VŨ QUỐC HOÀNG

BÍ KÍP LUYỆN
LẬP TRÌNH C

Quyển 1

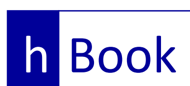


Bí kíp luyện Lập trình C Quyển 1

Bản cập nhật mới nhất của tài liệu này và các tài nguyên đính kèm được để ở: Facebook/vqhoang.books
Các ý kiến đóng góp xin gởi về: vqhoang.books@gmail.com
hoặc/và thảo luận tại: Facebook/vqhoang.books

Ông Nội Võ Văn Lầu

HSBN: 7966-7064-0406-070a-6f66-7d73-0205-0401
21-11-2017



Tưởng nhớ Nội, Ông Võ Văn Lầu

Lời nói đầu

Nếu bạn muốn dùng máy tính theo cách hay nhất, hãy học lập trình. Nếu bạn muốn học lập trình, hãy học lập trình C trước nhất. Nếu bạn muốn học lập trình C, hãy dùng tài liệu này. "*Bí kíp luyện Lập trình* C (*Quyển 1*)" được soạn chủ yếu cho việc tự học, phù hợp với đa số đối tượng, lứa tuổi và trình độ[1].

Cách hiệu quả nhất (và cũng là cách duy nhất) để học lập trình là *luyện lập trình*, nghĩa là học qua việc gõ, chạy, quan sát, phân tích, tìm hiểu, khám phá, sáng tạo, ... các chương trình. Tài liệu này được soạn theo phương pháp như vậy. Muốn giỏi lập trình, bạn phải luyện lập trình.

Thời gian phù hợp để bạn luyện xong Quyển 1 này là 16 tuần[2]: 3 tuần cho Tầng 1, 3 tuần cho Tầng 2, 4 tuần cho Tầng 3 và 6 tuần cho Tầng 4. Mỗi tuần nên luyện từ 2 đến 3 bài. Mỗi bài nên dành từ 1 giờ đến 3 giờ cho phần bài học, thêm 30 phút đến 1 giờ cho phần mở rộng (nếu có), sau đó nghỉ ngơi và làm bài tập[3]. Hiển nhiên là bạn phải luyện theo trình tự: từ tầng thấp lên tầng cao, từ bài đầu đến bài cuối[4].

Yêu cầu tiên quyết (và cũng là yêu cầu duy nhất) là bạn phải có máy tính. Đó phải là máy của riêng bạn để bạn có thể dùng mọi lúc mọi nơi[5]. Bạn đã sẵn sàng?[6] Hãy bắt đầu bằng việc cài đặt công cụ để lập trình (xem Phụ lục A.1) và luyện từ Bài 1.1. *Chúc bạn tu thành chính quả*!

Tài liệu này chắc chắn còn rất nhiều lỗi biên dịch và bug. Các ý kiến đóng góp xin gởi về địa chỉ mail vqhoang.books@gmail.com hoặc/và thảo luận tại Facebook/vqhoang.books

Cảm ơn một người. Nếu không có nhân duyên khơi nguồn thì đã không có tài liệu này.

Con cảm ơn Ba Me. Nhớ Nội...

Vũ Quốc Hoàng
Sài Gòn, 11-2017.

[1] Các cháu mẫu giáo, tiểu học không nên dùng tài liệu này:) Độ tuổi phù hợp: 15+. Trình độ tối thiểu: Toán lớp 8.

[2] Có thể tăng/giảm tùy theo quỹ thời gian và năng lực của bạn nhưng đừng luyện nhanh quá.

[3] Không nhất thiết phải làm hết các bài tập nhưng làm càng nhiều bài tập, công lực của bạn càng tăng nhanh.

[4] Luyện không theo trình tự có thể dẫn đến tẩu hỏa nhập ma:)

[5] 12 giờ khuya có vẻ là giờ linh:)

[6] Nghĩa là bạn muốn học (hoặc muốn thử học) lập trình C, bạn có chút ít thời gian rảnh và bạn đã có máy tính của riêng mình.

Mục lục

TẦNG 3

TẦNG 4

PHỤ LỤC

BÀI 1.1

Bạn hãy[1] gõ và chạy *chương trình C*[2] sau[3]. Lưu ý là gõ đúng y chang các kí tự kể cả xuống dòng và đúng chữ hoa chữ thường[4].

Mã 1.1.1[5] – Chương trình Hello World

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");

    return 0;
}
```

Nếu bạn làm đúng thì chương trình trên xuất ra dòng chữ[6]:

```
Hello World
```

Chúc mừng! Bạn vừa viết một chương trình C gởi lời chào đến toàn thể thế giới. Bạn có muốn gởi lời chào đến C không? Hãy sửa chương trình trên lại như sau.

Mã 1.1.2 – Chương trình Hello C

```
#include <stdio.h>
int main()
{
    printf("Hello C\n");

    return 0;
}
```

Và kết quả chạy là:

```
Hello C
```

[1] Khi tôi nói hãy làm gì đó thì bạn hãy làm thiệt, chứ không chỉ đọc suông nhé.
[2] C đọc là si hoặc xê, tốt nhất đọc là si, đừng đọc là cờ.
[3] Xem Phụ lục A.2 để biết cách "gõ và chạy chương trình C".
[4] C phân biệt chữ hoa chữ thường.
[5] Chỉ gõ văn bản ở cột bên phải, cột bên trái là số dòng để dễ theo dõi, màu sắc là để dễ nhìn.
[6] Xuất ra dòng chữ nghĩa là gì nhỉ? Lại xem Phụ lục A.2.

Bạn hãy so sánh hai chương trình để thấy chỗ tôi đã sửa. Bạn tên gì? Hãy[7] viết chương trình C gởi lời chào đến chính mình. Chẳng hạn chương trình của Dennis Ritchie[8] sẽ xuất ra dòng chữ sau:

```
Hello Dennis Ritchie
```

Bạn làm được không? Nếu được, chúc mừng, bạn có khiếu *lập trình C*[9] đó. Nếu không, hãy để ý hơn một chút để biết cách "*Bắt chước và Chỉnh sửa*". Nhắc lại: "Bắt chước và Chỉnh sửa". Đây là phương pháp cơ bản nhất để bạn bắt đầu học một kỹ năng nào đó, học lập trình C cũng vậy. Bạn được cho một *khuôn mẫu*, hãy quan sát để thấy đâu là phần cố định và đâu là phần thay đổi (hay chỗ trống), sau đó hãy chỉnh sửa phần thay đổi (hay điền vào chỗ trống) theo nhu cầu cụ thể của mình. Bạn chắc đã thấy mẫu đơn xin nghỉ học như hình dưới đây.

Hình 1.1.1 – Mẫu đơn xin nghỉ học

ĐƠN XIN NGHỈ HỌC

Kính gởi: - Ban giám hiệu nhà trường
- Giáo viên chủ nhiệm lớp

Em tên là: ....

Học lớp: ....

Em viết đơn này xin phép được nghỉ học ngày: ....

Lí do: ....

Kính mong các thầy cô xem xét, em hứa sẽ học bài và làm bài tập đầy đủ.

Em xin chân thành cảm ơn!

Nếu bạn cần nghỉ học thì bạn biết cách viết đơn cho mình thế nào rồi đó[10]. Thế thì khuôn mẫu của chương trình trên là gì?

Hình 1.1.2 – Mẫu chương trình Hello

```
#include <stdio.h>
int main()
{
    printf("Hello <tên>\n");

    return 0;
}
```

[7] Nhắc lại lần cuối nha: khi tôi nói hãy thì hãy làm chứ không chỉ đọc, đừng lười, cũng đừng cho rằng dễ quá mà bỏ qua, hãy ngoan ngoãn làm theo, ít nhất là ở Tầng 1 này:)

[8] Cha đẻ của ngôn ngữ C, xem Phụ lục A.3 để biết sơ về "Lịch sử C".

[9] Lập trình C tức là viết chương trình C.

[10] Hơi dư nhưng cũng phải chỉ lại cho chắc: copy mẫu đơn, điền tên mình, lớp, … vào chỗ trống thích hợp, in ra và nộp.

Chỗ được gạch dưới là chỗ cần sửa (hay cần điền) với ý nghĩa được nêu tương ứng. Phần còn lại là phần cố định (không sửa). Khuôn mẫu này là đúng, nhưng nếu như bạn muốn là Hi, Bonjour hay Chao (Chào) thay vì Hello thì sao? Khuôn mẫu tốt hơn là:

Hình 1.1.3 – Mẫu chương trình Hello phiên bản quốc tế:)

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("<dòng chữ cần xuất ra>\n");
5 
6     return 0;
7 }
```

Nếu bạn thấy được khuôn mẫu này thì bạn thực sự có khiếu đấy. Hãy chỉnh sửa (và xem kết quả) với khuôn mẫu này đi nhé[11]. Bạn có thể đi xa hơn nữa không? Sau đây là khuôn mẫu còn tổng quát hơn (mà nếu như bạn thấy được thì bạn đúng là thiên tài hoặc … bạn đã biết lập trình C rồi:)).

Hình 1.1.4 – Mẫu chương trình đơn giản

Mô tả bằng tiếng Việt	Mô tả bằng tiếng C
Đây là chương trình C của tôi. Hãy thực hiện các công việc sau đây cho tôi. Các công việc gồm: - Xuất ra dòng chữ: <dòng chữ 1> - Xuất ra dòng chữ: <dòng chữ 2> - … - Xuất ra dòng chữ: <dòng chữ *N*> Hết công việc. Kết thúc.	#include <stdio.h> int main() { printf("<dòng chữ 1>\n"); printf("<dòng chữ 2>\n"); … printf("<dòng chữ *N*>\n"); return 0; }

Tôi đã dùng cả tiếng Việt lẫn tiếng C (*ngôn ngữ C*) để mô tả khuôn mẫu trên. Thật vậy, một chương trình C có thể xem như một bài văn: một văn bản mô tả một nội dung nào đó bằng một ngôn ngữ nào đó. Như vậy việc *lập trình* (tức là viết chương trình) giống như việc viết văn và *lập trình viên* (tức là người viết chương trình) thì giống như nhà văn. Tuy nhiên, một chương trình C vẫn khác một bài văn tiếng Việt, chủ yếu ở hai chỗ:

- Nội dung: bài văn thường mô tả tri thức, tư tưởng, tâm tư, tình cảm, … của người viết còn chương trình mô tả các *công việc* mà *máy tính* (hoặc các dạng máy khác như smart-phone, …) cần phải làm.

[11] Bạn có thắc mắc về ý nghĩa của những từ ở các dòng khác Dòng 4 trong Hình 1.1.3 không, chẳng hạn `return 0;` ở Dòng 6 có nghĩa là gì? Tạm thời ta chưa cần hiểu hay để ý đến chúng. Đó là phần cố định, ta cứ để y nguyên là được. Nhiệm vụ của ta là tập trung vào phần thay đổi. Bạn chưa nên nhiều chuyện quá nhé:)

- Hình thức: bài văn tiếng Việt được viết, tổ chức theo qui định của tiếng Việt còn chương trình C được viết, tổ chức theo qui định của ngôn ngữ C.

Cũng lưu ý là cùng một nội dung lại có thể được mô tả bằng nhiều hình thức theo các ngôn ngữ khác nhau. Chẳng hạn cùng một nội dung là các công việc cần máy thực hiện, tôi đã mô tả nó bằng hai ngôn ngữ khác nhau là tiếng Việt và C. Cũng nội dung đó ta có thể mô tả bằng các ngôn ngữ khác như C++, C#, Java, …[12] và ta sẽ có các chương trình C++, C#, Java. Trong khi các ngôn ngữ như tiếng Việt, tiếng Anh, … được dùng phổ biến cho con người (và được gọi là *ngôn ngữ tự nhiên*) thì máy tính chưa có khả năng hiểu các ngôn ngữ này nên để mô tả các công việc mà máy cần thực hiện ta phải dùng các ngôn ngữ mà nó hiểu, gọi là các *ngôn ngữ lập trình*, như C, C++, C#, Java, … Điểm khác biệt quan trọng giữa ngôn ngữ lập trình với ngôn ngữ tự nhiên là ngôn ngữ lập trình rất đơn giản, ngắn gọn và rõ ràng[13]. Hơn nữa, ngôn ngữ lập trình thì dễ học hơn ngôn ngữ tự nhiên nhiều[14].

Với khuôn mẫu mới này ta có thể xuất ra nhiều dòng chữ thay vì chỉ một. Bạn hãy gõ và chạy chương trình sau.

Mã 1.1.3 – Chương trình Hello World trang trọng:)

```c
#include <stdio.h>
int main()
{
    printf("******************\n");
    printf("*   Hello World   *\n");
    printf("******************\n");

    return 0;
}
```

Chương trình trên xuất ra lời chào trang trọng được đóng khung, canh chỉnh trên nhiều dòng. Bạn hãy "Bắt chước và Chỉnh sửa" để đưa ra lời chào trang trọng đến mình.

Thật ra khuôn mẫu tổng quát hơn đó là:

Hình 1.1.5 – Mẫu chương trình

Đây là chương trình C của tôi. Hãy thực hiện các công việc sau đây cho tôi:
- <Công việc 1>
- <Công việc 2>
- …

[12] Các ngôn ngữ lập trình phổ biến khác.

[13] Ở điểm này thì một chương trình C không giống một bài văn (càng không giống một bài thơ hay bài nhạc, bức tranh) mà giống một lá đơn hay một lá thư hơn.

[14] Bạn có thể học ngôn ngữ C trong vòng 3 tháng nhưng phải mất nhiều chục năm để học tiếng Việt hay tiếng Anh.

- <Công việc *N*>

Với <Công việc> có thể là:

- Xuất ra dòng chữ: <dòng chữ cần xuất>
- … (Các dạng công việc khác sẽ học sau)

Như vậy một chương trình C yêu cầu máy tính thực hiện một *dãy các công việc*, theo thứ tự từ trên xuống, bắt đầu là Công việc 1, sau đó qua Công việc 2, … và kết thúc sau khi xong Công việc *N* (*N* hữu hạn). Có nhiều loại (kiểu, dạng) công việc khác nhau mà một trong số đó là việc xuất ra một dòng chữ. Các công việc cũng có mức độ đơn giản/phức tạp khác nhau. Công việc đơn giản thì được gọi là *lệnh*, chẳng hạn công việc xuất được gọi là *lệnh xuất*[15].

Bạn hãy nhớ khuôn mẫu trên mà ta có thể gọi là *mẫu chương trình* hay khung chương trình và là khuôn mẫu lớn nhất (có thể xem nó như mẫu bài văn). Còn nhiều khuôn mẫu nữa (nhỏ hơn có, lớn hơn có) mà ta sẽ gặp hay đã gặp như khuôn mẫu của việc xuất ra một dòng chữ (mà ta có thể coi như là một mẫu câu).

MỞ RỘNG 1.1 – Xuất tên tiếng Việt

Nếu tôi viết chương trình gởi lời chào đến mình thì tôi viết như sau[16].

Mã 1.1.4 – Chương trình Hello Hoàng

```
#include <stdio.h>
int main()
{
    printf("Ch\x85o Ho\x85ng\n");

    return 0;
}
```

Bạn hãy chạy thử để xem kết quả[17].

Bạn có thể xuất ra tên có dấu như tôi đã làm không? Có một vài chỗ bí hiểm ở dòng chữ cần xuất ra ở trên. Manh mối là *mã ASCII mở rộng*[18]. Nếu bạn không làm được thì tạm thời bỏ qua vụ có dấu nhé, bạn sẽ làm được khi luyện đến tầng cao hơn. Có ai đó nói rằng người Âu Mỹ lập trình C dễ hơn người Việt:) Cũng đúng vì máy tính ra đời ở Âu Mỹ và cha đẻ của C cũng là người Mỹ.

BÀI TẬP

[15] Lệnh (công việc đơn giản) có dấu hiệu nhận biết là thường được viết trên một dòng và kết thúc bằng dấu **;**

[16] Tôi tên Hoàng.

[17] Xem Phụ lục A.4 để biết cách cấu hình cửa sổ Console để hiển thị đúng kết quả.

[18] Đúng hơn là *trang mã* 437 (https://en.wikipedia.org/wiki/Code_page_437).

Bt 1.1.1 Viết chương trình C xuất ra[19]:

```
To C, or not to C: that is the question.
                 - William Shakespeare -
```

Bt 1.1.2 Viết chương trình C xuất ra (không bỏ dấu tiếng Việt)[20]:

```
   "Dưới cầu nước chảy trong veo
Bên cầu tơ liễu bóng chiều thướt tha."
              Truyện Kiều – Nguyễn Du.
```

Bt 1.1.3 Viết chương trình C xuất ra[21]:

```
    C (/ˈsiː/, as in the letter c) is a general-purpose,
imperative computer programming language, supporting
structured programming, lexical variable scope and
recursion, while a static type system prevents many
unintended operations. By design, C provides constructs that
map efficiently to typical machine instructions, and
therefore it has found lasting use in applications that had
formerly been coded in assembly language, including
operating systems, as well as various application software
for computers ranging from supercomputers to embedded
systems.
```

Bt 1.1.4 Viết chương trình C xuất ra “logo C” như mẫu[22]:

```
 CCCCC
CC    C
CC
CC
CC    C
 CccccC
  CCCC
```

[19] Một ví dụ hay trong sách “C Programming: A Modern Approach” của K. N. King. Chơi chữ từ câu thoại nổi tiếng “To be, or not to be: that is the question” trong vở kịch Hamlet của đại thi hào nổi tiếng người Anh William Shakespeare.
[20] Thử tìm cách xuất ra có dấu nháy kép "
[21] Giới thiệu ngôn ngữ lập trình C từ trang Wikipedia: https://en.wikipedia.org/wiki/C_(programming_language)
[22] Hãy sáng tạo thêm các mẫu khác và tạo logo cho mình.

BÀI 1.2

Các chương trình trong Bài 1.1 thật sự nhàm chán vì lần nào chạy nó cũng *thực thi*[1] và xuất ra cùng một kết quả. Bạn nghĩ sao nếu lần nào mở tivi lên ta cũng chỉ có thể coi mỗi một kênh, hay mở máy nghe nhạc lên mà lần nào nó cũng chơi cùng một bài nhạc. Ta, với vai trò là người dùng, mong đợi nhà sản xuất tạo ra các sản phẩm mà ta có thể điều chỉnh/tương tác khi sử dụng. Cái ti vi mà cho ta chọn kênh (một con số nguyên cho biết kênh số) hay cái máy nghe nhạc mà cho ta chọn bài hát (một chuỗi cho biết tên bài hát) thì tốt hơn[2]. Ở đây có sự tương tự mà ta cần lưu ý: chương trình là sản phẩm mà người lập trình chính là nhà sản xuất còn *người dùng chương trình* chính là người dùng sản phẩm[3].

Trở lại chương trình Hello (Mã 1.1.1), liệu ta có thể viết được một chương trình mà khi chạy, người dùng cho biết tên của mình và chương trình đưa ra lời chào đến người đó không? Hãy gõ và chạy chương trình sau.

Mã 1.2.1 – Chương trình Hello người dùng

```
#include <stdio.h>
int main()
{
    char name[100];

    printf("What's your name? ");
    gets(name);

    printf("Hello ");
    printf("%s", name);

    return 0;
}
```

[1] Sự thực thi của chương trình là toàn bộ việc thực hiện cùng kết quả xuất ra của chương trình, từ lúc bắt đầu đến khi kết thúc chương trình.

[2] Khi dùng cái bóng đèn thì ta muốn nó làm đúng mỗi một việc cố định nhàm chán là chiếu sáng:) Tuy nhiên, một cái bóng đèn cho phép ta điều chỉnh độ sáng (một giá trị thực) thì vẫn tốt hơn.

[3] Bạn phải phân biệt rõ ràng giữa người lập trình và người dùng vì bạn thường đóng cả hai vai trò, bạn viết chương trình xong, bạn chạy và dùng thử (thường là kiểm tra) nó.

Khi chạy, chương trình mời bạn nhập tên, sau khi bạn nhập tên của mình[4], chương trình xuất ra lời chào đến bạn. Chẳng hạn:

```
What's your name? Dennis Ritchie↲[5]
Hello Dennis Ritchie
```

Trước hết để ý rằng trong lệnh `printf("…");` ở Dòng 6 thiếu `\n` ở trước dấu " cuối. Bạn có thể thử thay lệnh đó bằng lệnh:

```
printf("What's your name? \n");
```

Chạy và quan sát kết quả sẽ thấy rằng trong lệnh `printf("…");` nếu có `\n` thì chuỗi xuất ra sẽ có xuống dòng còn không thì không có xuống dòng. Như vậy `\n` là *kí hiệu xuống dòng*.

Chương trình trên cũng có các *dòng trắng* (Dòng 5, 8, 11) mà mục đích là tách bạch các phần cho dễ đọc[6]. Vậy các phần đó là gì?

- Khai báo biến
- Thông báo nhập và nhập
- Xuất

Khi chương trình cho phép người dùng nhập dữ liệu thì ta cần cái gì đó trong chương trình để trữ dữ liệu, cái đó gọi là *biến*. Vì có nhiều kiểu (loại, dạng) dữ liệu như số nguyên, số thực, chuỗi, … nên ta cũng cần báo cho biết biến đó sẽ trữ dữ liệu kiểu gì. Ngoài ra ta cũng sẽ đặt tên cho biến để phân biệt các biến hay khi cần dùng đến biến. *Khai báo biến* chính là báo cho biết *tên biến* và *kiểu dữ liệu* mà biến sẽ chứa. Tên của biến nên gợi ý đến dữ liệu mà nó chứa (ý nghĩa, mục đích của dữ liệu đó, chẳng hạn như tên bài, kênh số, độ sáng). Ở đây biến chứa tên người dùng nên ta đặt tên là `name`[7]. Biến của ta có kiểu là chuỗi, vậy khuôn mẫu để khai báo *biến chuỗi* là:

char <tên biến chuỗi>[<số kí tự tối đa mà biến sẽ chứa>];

Con số 100 trong khai báo ở Dòng 4 cho thấy rằng tên mà người dùng nhập không được quá 100 kí tự. Để đề phòng trường hợp người dùng có tên dài hơn, bạn biết rồi đó, hãy nâng con số này lên.

---

[4] Xem Phụ lục A.2 để biết việc cách nhập Console.

[5] Chữ gạch dưới là chữ mà người dùng nhập, ↲ là kí hiệu của phím Enter.

[6] Bạn có thể bỏ hoặc thêm các dòng trắng tùy thích, mà mục đích là để viết chương trình cho đẹp.

[7] Cũng nên đặt tên biến ngắn gọn để đỡ công gõ phím. Ta cũng thường dùng từ tiếng Anh để đặt tên biến vì nó ngắn gọn và rõ ràng hơn tiếng Việt. Ở đây, `name` (tiếng Anh) nghĩa là tên. Ta có thể đặt tên biến là `ten` (ý là tên) nhưng như vậy thì khó hiểu quá. Lưu ý là bạn chưa được phép dùng tiếng Việt có dấu để đặt tên biến.

Trong phần thông báo nhập và nhập thì lệnh: `gets(name)`[8] ở Dòng 7 đợi người dùng nhập vào một chuỗi, sau khi người dùng nhập chuỗi và nhấn Enter thì chuỗi sẽ được đưa vào trữ trong biến `name`. Như vậy khuôn mẫu nhập một chuỗi vào một biến chuỗi là:

```
gets(<tên biến chuỗi>);
```

Trước khi chương trình đợi người dùng nhập thì chương trình đưa ra thông báo cho biết là đợi người dùng nhập tên bằng lệnh: `printf("What's your name? ")`. Nếu không có thông báo này thì người dùng không biết chương trình đang đợi mình và đợi mình nhập cái gì. Như vậy nên có thông báo rõ ràng để người dùng dễ dàng dùng chương trình. Việc chiều chuộng người dùng là điều dễ hiểu, nhất là khi chương trình là các sản phẩm thương mại, nơi mà khách hàng là thượng đế. Thực sự, việc chương trình dễ dùng với người dùng[9] là một tiêu chí quan trọng để đánh giá chương trình, nhất là các chương trình thương mại.

Phần xuất thì xuất *chuỗi cố định* bằng lệnh `printf("Hello ")` (lưu ý là xuất ra chuỗi mà không xuống dòng, không có kí hiệu `\n`) và sau đó là xuất ra chuỗi chứa trong biến `name` (là chuỗi mà người dùng đã nhập vào trước đó) bằng lệnh `printf("%s", name)`. Bạn có thấy khuôn mẫu của việc xuất ra chuỗi chứa trong một biến chuỗi nào đó không?[10]

```
printf("%s", <tên biến chuỗi>);
```

Có một điều quan trọng chưa được nhắc đến. Thực ra, khi người dùng cung cấp (nhập) tên bài hát, máy nghe nhạc phải xử lý để tìm bài hát tương ứng rồi mới phát (xuất) bài hát đó. Các chương trình cũng vậy, sau khi nhận dữ liệu từ người dùng phải xử lý để có dữ liệu mong muốn rồi mới xuất ra. Khi dùng một Calculator (máy tính cầm tay) bạn nhập dữ liệu (các con số và phép tính), máy tính toán rồi xuất ra kết quả. Sau đây là một chương trình như vậy.

Mã 1.2.2 – Chương trình Calculator đơn giản:)

```c
#include <stdio.h>
int main()
{
    int a;
    int b, tong, tich;

    printf("a = ? ");
    scanf("%d", &a);
    printf("b = ? ");
```

[8] Tôi đã không viết dấu `;` ở đây nhưng bạn phải gõ nó trong chương trình.

[9] Thuật ngữ là *thân thiện người dùng*.

[10] Kể từ đây bạn tự phát hiện ra những khuôn mẫu đơn giản, lặt vặt như vậy nhé.

```
10      scanf("%d", &b);
11
12      tong = a + b;
13      tich = a * b;
14
15      printf("Tong la: ");
16      printf("%d", tong);
17      printf("\nTich la: %d", tich);
18
19      return 0;
20  }
```

Chương trình trên cho người dùng nhập 2 con số nguyên, tính rồi xuất ra tổng và tích của hai số đó. Chẳng hạn:

```
a = ? 10↲
b = ? 20↲
Tong la: 30
Tich la: 200
```

Ở Dòng 4 ta khai báo một biến tên `a` dự định chứa số nguyên mà người dùng nhập. Như vậy khuôn mẫu khai báo *biến kiểu nguyên* là:

int <tên biến nguyên>;

Tương tự ta có thể khai báo các biến nguyên `b`, `tong`, `tich` (mà mục đích là chứa số thứ hai, tổng và tích) bằng 3 dòng khai báo nữa, nhưng Dòng 5 cho thấy cách *khai báo gộp*, ngắn gọn hơn bằng cách dùng dấu phẩy phân cách các tên biến. Như vậy, bạn biết cách khai báo gọn 10 biến cùng kiểu nguyên rồi đó[11]. Hiện tượng có những cách viết tắt, ngắn gọn hơn cho cùng một nội dung xuất hiện trong nhiều ngôn ngữ mà ngôn ngữ C không là ngoại lệ.

Dòng 7 là thông báo nhập. Dòng 8 là lệnh nhập như ở Dòng 7 Mã 1.2.1 nhưng là nhập một số nguyên vào biến nguyên. Khuôn mẫu là:

`scanf("%d", &`<tên biến nguyên>`);`

Các Dòng 9, 10 vận dụng cùng khuôn mẫu để thông báo và nhập cho biến nguyên `b`.

Dòng 12 và 13 cho thấy một loại lệnh mới mà được xem là quan trọng nhất trong các chương trình C, đó là *lệnh gán* với khuôn mẫu:

<tên biến> = <biểu thức>;

Lệnh gán tính giá trị của *biểu thức* và đưa giá trị đó vào trữ trong biến có tên tương ứng. Dữ liệu cũ trong biến sẽ không còn nữa, biến sẽ chứa giá trị

[11] Dùng 9 dấy phẩy phân cách 10 tên biến.

mới[12]. Ta sẽ tìm hiểu kĩ hơn biểu thức ở các bài sau. Tạm thời thì đó là các công thức tính toán mà ta đã gặp trong Toán phổ thông. Như vậy lệnh gán ở Dòng 12 tính tổng của a và b rồi bỏ vào (gọi là *gán*) biến tong. Biến tong như vậy chứa tổng hai số nguyên mà người dùng đã nhập. Lệnh gán ở Dòng 13 thì tính tích của a, b rồi gán vào biến tich. Bạn đang phân vân? Đúng vậy, trong khi dấu + kí hiệu cho *phép cộng* là quen thuộc thì dấu * kí hiệu cho *phép nhân* là hơi lạ. Ta thường dùng dấu × để kí hiệu cho phép nhân, nhưng vì bàn phím không gõ được dấu này[13]. Lưu ý là ta cũng hay dùng dấu . kí hiệu cho phép nhân nhưng C lại dùng nó cho mục đích khác mà hồi sau ta sẽ rõ.

Bạn đã biết cách xuất ra chuỗi chứa trong một biến chuỗi (Dòng 10 Mã 1.2.1). Còn xuất ra số nguyên chứa trong một biến nguyên? Lệnh xuất ở Dòng 16 cho thấy khuôn mẫu:

```
printf("%d", <tên biến nguyên chứa giá trị cần xuất>);
```

Tương tự bạn có thể dùng hai lệnh xuất (như Dòng 15 với Dòng 16) để xuất ra tích của a và b. Nhưng lệnh xuất ở Dòng 17 một lần nữa cho thấy tính ngắn gọn của ngôn ngữ. Hãy suy nghẫm một chút nếu bạn chưa thấy khuôn mẫu nhé. Bạn có thể gom 3 lệnh xuất ở Dòng 15, 16, 17 vào một lệnh xuất duy nhất mà vẫn cho cùng kết quả không?[14]

Cũng lưu ý: biến a, b chứa dữ liệu nhập từ người dùng nên có thể gọi là *biến nhập*, nhưng biến tong, tich thì không chứa dữ liệu nhập mà chứa kết quả tính toán nên có thể gọi là *biến tạm*. Bạn cũng có thể nói tong, tich là *biến xuất* vì ta đã dùng nó trong lệnh xuất. Tuy nhiên gọi như vậy là không đáng vì ta có thể không cần nó mà vẫn tính và xuất ra được tổng (tương ứng cho tích) bằng lệnh:

```
printf("Tong la: %d", a + b);
```

Như vậy khuôn mẫu để tính và xuất ra giá trị nguyên mà không cần dùng biến tạm là:

```
printf("...%d...", <biểu thức với giá trị cần xuất>);
```

Bạn hãy vận dụng các điều đã học đến đây để viết lại chương trình trên thật ngắn gọn nhé.

Với các biến, một cách trực quan, ta có thể xem chúng như là những cái hộp chứa dữ liệu, mà bên ngoài hộp ghi thông tin của biến gồm tên biến và kiểu dữ liệu mà nó chứa, còn bên trong hộp thì chứa dữ liệu của biến. Ta có thể đưa dữ liệu vào trong hộp qua lệnh nhập hoặc lệnh gán và đọc (xem) dữ liệu trong hộp qua tên biến trong biểu thức. Cũng lưu ý, hộp chỉ chứa một

---

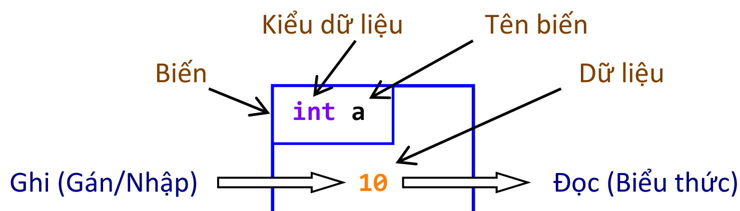
[12] Do đó nó có tên là biến tức là có thể thay đổi.

[13] Trông nó giống chữ x nhưng không phải nhé.

[14] Đáp án: `printf("Tong la: %d\nTich la: %d", tong, tich);`

dữ liệu tại một thời điểm nhưng có thể thay đổi dữ liệu qua các thời điểm khác nhau.

Hình 1.2.1 – Biến như là "hộp chứa dữ liệu"



Chương trình trên mặc dù đơn giản nhưng cho ta khuôn mẫu phổ biến của một chương trình C:

Hình 1.2.2 – Mẫu chương trình C phổ biến (Nhập – Xử lý – Xuất)

Đây là chương trình C của tôi.
Tôi có các biến sau đây:
- <Các khai báo biến>

Hãy thực hiện các công việc sau đây cho tôi:
- <Nhập dữ liệu từ người dùng cho các biến>
- <Xử lý/Tính toán>
- <Xuất dữ liệu cho người dùng>

Bạn đã biết cách tính tổng và tích của hai số nguyên. Còn tính hiệu hai số nguyên thì sao? Dùng dấu – cho phép trừ thôi mà. Còn phép chia hai số nguyên thì sao? Cũng rắc rối à: 6 kg gạo đem chia cho 4 người thì mỗi người được 1 kg rưỡi (1.5 kg) nhưng 6 cây kẹo đem chia cho 4 người thì mỗi người được 1 cây và dư 2 cây. Vậy thì 6 chia cho 4 được 1.5 hay được 1 dư 2? Trả lời: cả hai cách chia đều đúng, cách thứ nhất là *chia theo kiểu thực* còn cách thứ hai là *chia theo kiểu nguyên*. Trong C, khi cả số bị chia (6) lẫn số chia (4) đều là số nguyên thì phép chia được thực hiện theo kiểu nguyên, kết quả ta được thương (1) và số dư (2) cũng là các số nguyên. Ta dùng dấu / để tính thương và dấu % để tính số dư. Như vậy 6/4 được 1 và 6%4 được 2.

Phép chia còn rắc rối hơn khi dính đến số âm. -6 chia cho 4 được thương và số dư bao nhiêu? Để dễ hình dung, hãy chia số bị chia a cho số chia b theo kiểu thực, sau đó lấy *phần nguyên* của số thực[15] thì được thương q, sau đó tính phần dư $r = a - b \times q$. Chẳng hạn chia $a = 6$ cho $b = 4$ theo kiểu thực được 1.5, lấy phần nguyên được thương là $q = 1$, tính số dư $r = 6 - 4 \times 1 = 2$. Tương tự, chia $a = -6$ cho $b = 4$ theo kiểu thực được -1.5, lấy

[15] Phần nguyên của một số thực là phần trước dấu chấm thập phân và *phần lẻ* là phần sau dấu chấm thập phân, chẳng hạn 1.5 có phần nguyên là 1 và phần lẻ là 0.5.

phần nguyên được thương là q = -1, tính số dư r = -6 – 4 × (-1) = -2[16]. Đố bạn 6 chia cho -4, -6 chia cho -4 được thương và số dư bao nhiêu?[17] Cũng nhận xét là số dư r luôn có cùng dấu với số bị chia a. Như vậy, lưu ý, một số nguyên a là chẵn trong C nếu `a % 2` là 0 còn a là lẻ nếu `a % 2` khác 0 (chứ không phải là 1 vì nó có thể là 1 nếu a dương và là -1 nếu a âm).

Một điều quan trọng nữa về phép chia: ta *không thể chia cho 0*. Nếu một chương trình C, khi chạy, xảy ra trường hợp chia cho 0 (bằng dấu / hay %) thì chương trình đó sẽ bị lỗi[18] và dừng (kiểu như bạn đang xem tivi thì nó chớp chớp, giựt giựt rồi tắt ngúm. Bạn chỉ có nước rút điện, để một thời gian rồi mở lại). Tuy nhiên ta có thể chia 0 cho mọi số (khác không), được thương là 0 và số dư là 0.

MỞ RỘNG 1.2 – Tràn số nguyên

Bạn hãy dọc chương trình tính tổng và tích trên (Mã 1.2.2) (nghĩa là bạn chạy nó nhiều lần với các giá trị nhập khác nhau). Có khi nào kết quả tính ra có vấn đề (nghĩa là không đúng) không? Bạn thử nhập a là 2000000000 (2 tỉ), `b` là 2000000000 (2 tỉ) thì được tổng là -294967296. Rõ ràng là kết quả tính ra không đúng. Chương trình có vấn đề. Tuy nhiên vấn đề không phải do ta viết chương trình sai và cũng không phải do chương trình chạy sai. Vấn đề là số nguyên (kiểu dữ liệu `int`) trong C không phải là số nguyên trong Toán.

Trong Toán tập số nguyên là vô hạn, không có số nhỏ nhất, cũng không có số lớn nhất. Nhưng tập *số nguyên của C* là hữu hạn, có số lớn nhất và số nhỏ nhất. Số nhỏ nhất là -2147483648 và số lớn nhất là 2147483647. Không có số nguyên nào của C nằm ngoài *phạm vi* này. Do đó nếu kết quả đúng cần tính ra (không chỉ là cộng mà tất cả các phép tính khác) nằm ngoài phạm vi này thì ta sẽ được kết quả thực tế nằm trong phạm vi nên kết quả không còn đúng nữa. Hiện tượng này được gọi là *tràn số*. Chẳng hạn, kết quả đúng của 2 tỉ + 2 tỉ cần tính ra phải là 4 tỉ thì ta lại được kết quả thực tế không đúng là -294967296.

Tại sao các số nguyên trong C (hay trên máy) là hữu hạn? Có thể hiểu được vì máy dùng các ô nhớ để lưu trữ chúng (chính là các ô nhớ của biến tương ứng), mà số ô nhớ là hữu hạn. Tại sao giá trị nhỏ nhất là -2147483648 (là -2^{31}), giá trị lớn nhất là 2147483647 (là $2^{31} – 1$)? Hay tại sao 2 tỉ + 2 tỉ lại được -2147483648? Để trả lời, ta cần phải đi chi tiết vào *cách thức tổ chức và lưu trữ* số nguyên trên máy. Làm sao để giải quyết vấn đề tràn số để cho kết quả tính toán đúng? Ta sẽ tìm hiểu thêm những

[16] Trong Toán cao cấp thì -6 chia cho 4 lại được -2 và dư 2.
[17] Đáp án: lần lượt là q = -1, r = 2 và q = 1, r = -2.
[18] Lỗi này được gọi là lỗi run-time mà ta sẽ tìm hiểu sau.

điều này sau.

Câu hỏi quan trọng khác: làm sao biết là đã tràn số? Là khi kết quả thực tế tính ra không giống với kết quả đúng cần tính ra hay khi kết quả đúng cần tính ra nằm ngoài phạm vi. Nhưng ta đâu có biết kết quả đúng cần tính ra:(Mẹo đơn giản là: nếu có *bất thường về dấu* của kết quả thì có tràn số. Chẳng hạn: cộng hai số dương lại được số âm (như cộng 2 tỉ với 2 tỉ lại được số âm -2147483648), trừ số dương cho số âm lại được số âm hay nhân hai số dương lại được số âm, … Tuy nhiên, có phải mọi tràn số đều dẫn đến bất thường về dấu hay có trường hợp tràn số nào mà không phát hiện được bằng dấu không? Hồi sau ta sẽ rõ:)

BÀI TẬP

Bt 1.2.1 Viết chương trình cho nhập <name> rồi xuất ra bài hát "Happy birthday":

```
Happy birthday to <name>.
Happy birthday to <name>.
Happy birthday, happy birthday, happy birthday to <name>.
```

Bt 1.2.2 Viết chương trình cho nhập tên <Roméo> và <Juliet> rồi xuất ra đoạn trích[19]:

```
"... <Roméo> bước thơ thẩn trong vườn, tơ tưởng đến <Juliette>. Bỗng nhiên, một cánh cửa sổ từ từ hé mở, <Juliette> hiện ra, tựa vào bao lơn. Cô cũng bồn chồn và lo lắng, rồi trong khi <Roméo>, ẩn mình trong bóng tối, so sánh nàng với bình minh và khung cửa với phương Đông, chế nhạo mặt trăng mờ nhạt vì hờn ghen với nhan sắc kiều diễm của <Juliette> ..."
```

Bt 1.2.3 Viết chương trình cho nhập vào năm sinh của một người và xuất ra tuổi của người đó.

Bt 1.2.4 Viết chương trình cho nhập số nguyên dương có 3 chữ số. Xuất ra tổng các chữ số của số đó. *Ví dụ*: nhập số 123, xuất ra 6. (Vì 1 + 2 + 3 = 6).

Bt 1.2.5 Viết chương trình cho nhập một số nguyên và xuất ra trị tuyệt đối của số đó. *Ví dụ*: nhập số -4, xuất ra 4. (Vì |-4| = 4). (*Gợi ý*: thử dùng phép chia lấy dư cho 2).

Bt 1.2.6 Viết chương trình thử nghiệm với phép chia nguyên /, %. Nhập các giá trị khác nhau, tự tính tay và so kết quả tính ra với chương trình.

---

[19] Lượm lặt đâu đó trên mạng.

BÀI 1.3

Bạn hãy gõ và chạy chương trình sau.

Mã 1.3.1 – Chương trình tính chu vi và diện tích hình tròn

```
1  #include <stdio.h>
2  int main()
3  {
4      double r, c, s;
5  
6      printf("r = ? ");
7      scanf("%lf", &r);
8  
9      c = 2 * 3.1416 * r;
10     s = 3.1416 * r * r;
11 
12     printf("Chu vi: %lf.\nDien tich: %lf.\n", c, s);
13 
14     return 0;
15 }
```

Chương trình trên cho người dùng nhập bán kính của một hình tròn, tính và xuất ra chu vi, diện tích của hình tròn đó theo công thức:

Chu vi = $2\pi r$ và *Diện tích* = πr^2 với *r* là bán kính hình tròn.

Số π (pi) có giá trị gần đúng là 3.1416[1].

Chương trình trên cũng cùng khuôn mẫu như Bài 1.2 (khai báo biến, nhập, tính toán và xuất). Điểm khác là ta làm việc với *số thực* (hiểu nôm na là số có phần lẻ, chẳng hạn như số π).

- Trong khai báo biến thực ta thay từ `int` bằng từ `double`: chúng còn được gọi là *từ khóa*[2] mà ý nghĩa là cho biết kiểu dữ liệu: `int` là kiểu số nguyên, `double` là kiểu số thực. Ở Dòng 4 ta đã khai báo 3 biến kiểu thực (`double`) là `r`, `c` và `s`[3].
- Trong lệnh nhập, ta thay `%d` bằng `%lf`: chúng được gọi là *chuỗi định dạng*[4] mà `%d` là cho số nguyên và `%lf` là cho số thực[5]. Tương tự,

---

[1] Dùng dấu chấm (.) theo kiểu Mỹ chứ không phải dấu phẩy (,) theo kiểu Việt Nam.

[2] Từ khóa là từ mang nghĩa đặc biệt mà ta sẽ rõ trong bài sau.

[3] Ta cũng thường đặt tên biến ngắn gọn như kí hiệu trong Toán. Trong Toán thì bán kính hay kí hiệu là *r* (viết tắt của radius), chu vi là *c* (viết tắt của circumference hay chu vi:)) và diện tích là *s* (lại là viết tắt của area:)).

[4] Ta lại sẽ rõ trong bài sau.

trong lệnh xuất, `%lf` được dùng để xuất giá trị thực. Dòng 7 và Dòng 12 cho thấy điều này.

Có hai điều quan trọng nữa. Đầu tiên là lệnh gán ở Dòng 9. Ta đã biết rằng bên phải của lệnh gán là biểu thức mà giá trị của nó sẽ được tính và gán vào biến. So với bài trước thì biểu thức này phức tạp hơn: nó có nhiều phép toán hơn. Biểu thức trong trường hợp đơn giản nhất là *hằng số* (nguyên hoặc thực như 2 hay 3.1416) mà giá trị chính là hằng số đó hay là *biến* mà giá trị là giá trị đang chứa trong biến đó. Trường hợp phức tạp hơn thì biểu thức được tạo nên từ các biểu thức nhỏ hơn (gọi là các *toán hạng*) và các *toán tử* (các dấu +, -, *, /, %, …). Các toán tử kí hiệu cho các *phép toán*, là các thao tác tính toán tương ứng: + kí hiệu cho thao tác cộng, - kí hiệu cho thao tác trừ, … Giá trị của *biểu thức phức* (tạp) được xác định từ giá trị của các toán hạng và các phép toán tương ứng[6].

Khi có nhiều toán tử thì ta có thể dùng cặp ngoặc tròn để xác định phép toán nào được thực hiện trước hoặc dựa theo *độ ưu tiên* của các toán tử và *tính kết hợp* mà bạn đã biết từ lâu: "Nhân chia trước, cộng trừ sau, nếu cùng (cộng/trừ hoặc nhân/chia) thì từ trái qua phải". Chẳng hạn biểu thức `(1 + 2)*(3 + 4)` có giá trị là 21 khác với `(1 + 2)*3 + 4` có giá trị là 13, hay `1 + 2*3 + 4` có giá trị là 11. Còn như biểu thức `1 – 2 – 3` có giá trị là -4, trong khi `1 – (2 – 3)` có giá trị là 2.

Bất kì chỗ nào cần giá trị (hay dữ liệu[7]), ta mô tả nó bằng một biểu thức. Khi chạy chương trình, biểu thức luôn được tính (còn gọi là *lượng giá*) ra giá trị (hay dữ liệu), rồi giá trị đó được chương trình dùng trong những bước tiếp theo. Như vậy biểu thức là phương tiện của ngôn ngữ C dùng để mô tả nội dung là giá trị. Chẳng hạn khuôn mẫu đúng của lệnh xuất một giá trị thực là:

```
printf("...%lf...", <giá trị thực cần xuất>);
```

Như vậy bạn có thể xuất ra giá trị của hằng số π bằng lệnh `printf("%lf", 3.1416)` hay xuất ra giá trị chứa trong biến thực `s` bằng lệnh `printf("%lf", s)` hay xuất ra diện tích của hình tròn với bán kính chứa trong biến thực `r` bằng lệnh `printf("%lf", 3.1416 * r * r)` mà không cần dùng biến tạm. Trong 3 trường hợp trên thì biểu thức lần lượt là hằng, biến và biểu thức phức.

Ta cần phân biệt biểu thức với lệnh: về hình thức, biểu thức như cụm từ còn lệnh như là câu; về nội dung, biểu thức mô tả giá trị còn lệnh mô tả công việc. Biểu thức là *đơn vị tính toán* trong khi lệnh là *đơn vị thực thi* của

---

[5] d là viết tắt của decimal, `lf` là viết tắc của long floating-point.
[6] Bạn sẽ học đầy đủ hơn về biểu thức trong bài sau.
[7] Trường hợp dữ liệu là số thì hay được gọi là giá trị.

chương trình. Thường thì biểu thức là một phần tạo nên lệnh, như trong lệnh gán thì khuôn mẫu là:

<tên biến> = <biểu thức>;

Điều quan trọng nữa của chương trình trên là lệnh xuất ở Dòng 12. Trong Bài 1.2 bạn đã được nghe đến tính cô đọng (gọn gàng) của ngôn ngữ. Ở đây nó còn dữ dội hơn. Nếu không dùng lệnh xuất ở Dòng 12, bạn sẽ phải dùng hai lệnh xuất:

```
printf("Chu vi: %lf.\n", c);
printf("Dien tich: %lf.\n", s);
```

Bạn có thấy được khuôn mẫu của việc rút gọn này không?

Hình 1.3.1 – Mẫu xuất cô đọng

```
printf("Chu vi: %lf.\nDien tich: %lf.\n", c, s);
```

Nhớ "*thứ tự rất quan trọng*" nhé!

Để thấy được khuôn mẫu mới này bạn cần làm nhiều hơn là "Bắt chước và Chỉnh sửa". Từ "Bắt chước và Chỉnh sửa" các khuôn mẫu của lệnh xuất, từ phân tích về tính cô đọng của ngôn ngữ và từ quan sát các khuôn mẫu khác như khuôn mẫu khai báo gộp nhiều biến, mà bạn mở rộng, phát triển được khuôn mẫu mới này. Đây chính là phương pháp học ở mức kế tiếp mà bạn cần nắm vững: "*Bắt chước – Chỉnh sửa – Phân tích – Phát triển*". Trong khi "Bắt chước – Chỉnh sửa" chỉ là phát hiện khuôn mẫu (thường được cho hoặc dễ thấy) và điền vào chỗ trống thì "Phân tích – Phát triển" đòi hỏi phát hiện khuôn mẫu mới, mở rộng hơn, tổng quát hơn từ nhiều khuôn mẫu và qui tắc khác. Bí quyết là: quan sát cái mới, so sánh với cái cũ, nhận ra cái tương đồng và khác biệt. Hơn nữa, bạn có thể cần phải "*Thử sai*" để kiểm tra những giả thuyết, phỏng đoán rút ra là đúng hay sai. Chẳng hạn từ khuôn mẫu của khai báo, khuôn mẫu của lệnh gán và từ nguyên lý cô đọng của ngôn ngữ, liệu ta có thể kết hợp:

```
double c;
c = 2 * 3.1416 * r;
```

thành một thứ gọn hơn (mà ta cũng không biết gọi là gì, "khai báo kết hợp gán" chăng?[8]):

```
double c = 2 * 3.1416 * r;
```

[8] Gọi là *khai báo với khởi tạo*.

Phỏng đoán này có vẻ hợp lý. Để biết rằng nó có được C chấp nhận hay không thì bạn hãy "*Thử sai*" để kiểm tra bằng cách chạy thử. Eureka! (Hoan hô!) Nó chạy được.

Bạn có thể viết thêm vào chương trình trên để tính thể tích của khối cầu có bán kính *r* theo công thức:

Thể tích = $(4/3)\pi r^3$.

Tôi cá là bạn sẽ làm sai[9]. Có phải bạn đã khai báo thêm biến thực v[10] và tính thể tích bằng lệnh gán:

```
v = 4/3 * 3.1416 * r*r*r;
```

Nếu bạn kỹ hơn sẽ viết:

```
v = (4/3) * 3.1416 * r*r*r;
```

Hai cách viết trên là như nhau, có thể bỏ cặp ngoặc tròn đi vì cả hai biểu thức đều được tính bằng cùng một cách và cho ra cùng kết quả. Mọi thứ trông có vẻ tốt[11], nhưng … Sai! Bạn sai ở chỗ bạn mong đợi kết quả của `4/3` là 1.333333… nhưng thực tế C lại tính ra `4/3` là 1. Bạn đã biết điều này ở bài trước: nếu cả số bị chia (4) lẫn số chia (3) là số nguyên thì phép chia là chia nguyên và toán tử `/` cho phần thương nguyên (là 1 thay vì 1.33333…). Về mặt kỹ năng lập trình thì đây có vẻ là lỗi nhỏ do sơ suất nhưng hậu quả thì có thể rất lớn. Thử tưởng tượng, biểu thức này (hay các biểu thức tương tự) được dùng để tính thể tích của máy bay. Kết quả là ta chế ra chiếc máy bay có thể tích nhỏ hơn cần thiết (chỉ bằng 3/4 mong đợi) và dẫn đến một thảm họa hàng không:) Những lỗi như thế này được gọi là *lỗi logic*[12], nó cực kì khó phát hiện vì không để lại dấu vết gì. Trong C rất dễ mắc những lỗi như vậy. Hãy cẩn thận!

Sửa lỗi này bằng cách nào? Rất đơn giản, hãy để C thực hiện phép chia theo kiểu thực: 4.0 / 3.0 được chia theo kiểu thực và cho kết quả mong đợi là 1.333333…. Hãy sửa lại là:

```
v = 4.0/3.0 * 3.1416 * r*r*r;
```

Tuy nhiên: 4.0 / 3 hay 4 / 3.0 cũng được chia theo kiểu thực và cho kết quả là 1.33333… Trong C, ta cũng thường thấy các biểu thức có "*kiểu hỗn tạp*" như vậy, là biểu thức có chứa các hằng số hay biến có kiểu khác nhau (thường là các kiểu số như số nguyên, số thực). Trong trường hợp như vậy, C sẽ tự động (và âm thầm) chuyển kiểu dữ liệu trước khi thực hiện phép

[9] Xin lỗi đã không tin tưởng bạn.

[10] Là kí hiệu Toán hay dùng cho thể tích (viết tắt của volume).

[11] Đừng nói là bạn không biết tính r^3 bằng `r*r*r`. "Bắt chước – Chỉnh sửa – Phân tích – Phát triển" từ r^2 thôi mà.

[12] Ta sẽ tìm hiểu thêm về các lỗi và cách giải quyết chúng trong các bài sau.

toán và thường giá trị nguyên sẽ được chuyển thành giá trị thực nếu cần. Chẳng hạn để tính 4 / 3.0 thì C sẽ chuyển tự động giá trị 4 (nguyên) thành 4.0 (thực) rồi thực hiện phép chia thực 4.0 cho 3.0 được 1.33333…

Trong khi việc *chuyển tự động* giá trị nguyên sang giá trị thực khi cần là phổ biến và an toàn (giá trị 4 nguyên chuyển thành 4.0 thực không mất mát gì cả) thì việc chuyển tự động giá trị thực sang giá trị nguyên cũng có thể xảy ra và đáng lưu ý. Ví dụ với khai báo và khởi tạo sau:

```
int a = 2.001, b = 2.499, c = 2.500, d = 2.999;
```

thì cả 4 biến nguyên `a`, `b`, `c`, `d` đều có giá trị nguyên là 2. C đã chuyển tự động các giá trị thực 2.001, 2.499, 2.500, 2.999 thành giá trị nguyên và đều được giá trị là 2. Qui tắc là: C sẽ *cắt bỏ phần lẻ* (chứ không làm tròn) khi chuyển giá trị thực thành giá trị nguyên.

Bạn cũng phải lưu ý đến thứ tự thực hiện các phép toán để đảm bảo việc chuyển kiểu là đúng đắn. Chẳng hạn với lệnh gán: `v = 4/3 * 3.1416 * r*r*r` thì ta sẽ được giá trị sai vì C sẽ tính `4/3` trước (rồi mới thực hiện các phép nhân sau theo qui tắc kết hợp từ trái qua phải) mà phép chia này là chia nguyên (do cả 4 lẫn 3 đều là các giá trị nguyên). Nhưng với lệnh gán: `v = 4/(3*1.0) * 3.1416 * r*r*r` thì ta sẽ được giá trị đúng. C sẽ tính `3*1.0` trước (do cặp ngoặc tròn) mà khi đó giá trị nguyên 3 sẽ được chuyển tự động thành giá trị thực 3.0 rồi thực hiện phép nhân được kết quả là giá trị thực 3.0. Sau đó C tính 4/3.0 thì sẽ tự động chuyển giá trị nguyên 4 thành giá trị thực 4.0 rồi thực hiện phép chia thực 4.0/3.0. Dĩ nhiên ta có thể viết đơn giản hơn: `v = 4/3.0 * 3.1416 * r*r*r`.

Ta thường muốn chuyển giá trị thực thành giá trị nguyên bằng cách làm tròn thay vì cắt bỏ phần lẻ. Có một mẹo đơn giản. Ví dụ, muốn chuyển giá trị thực trong biến thực `r` thành giá trị nguyên bằng cách làm tròn và bỏ vào biến nguyên a thì ta có thể viết như sau:

```
a = r + 0.5;
```

Nếu `r` chứa 2.001 hay 2.499 thì a sẽ chứa giá trị nguyên 2. Nếu `r` chứa 2.500 hay 2.999 thì a sẽ chứa giá trị nguyên 3[13].

Còn như bạn muốn lấy phần lẻ của một giá trị thực (như 2.499 có phần lẻ là 0.499) thì làm sao? Ví dụ sau đây lấy phần lẻ của giá trị thực trong biến thực `r` bỏ vào biến thực `b`: `b = r - (int)r;`[14]

MỞ RỘNG 1.3 – Tràn số thực

[13] Hãy suy ngẫm một chút để hiểu tại sao.

[14] `(int)r` sẽ bắt buộc C chuyển giá trị thực chứa trong `r` thành giá trị nguyên và do đó được phần nguyên của `r`. Ta sẽ tìm hiểu kĩ hơn sau.

Bạn có biết: $0.1^2 = 0.01$. Đây hiển nhiên là chân lý rồi. Tuy nhiên, trong C, nó không còn đúng nữa. Hãy chạy chương trình sau.

Mã 1.3.2 – Chương trình tính 0.1^2

```
#include <stdio.h>
int main()
{
    printf("%.20lf\n", 0.1*0.1);

    return 0;
}
```

Kết quả cho thấy $0.1^2 \neq 0.01$!

Dĩ nhiên là chân lý không sai. Mà C cũng không sai. Vấn đề là *số thực trong C* (kiểu `double`) không phải là số thực trong Toán. Trong Toán tập số thực là vô hạn, và phần lẻ (phần sau dấu chấm thập phân) của một số thực nào đó cũng có thể vô hạn như số 1/3 = 0.333333... . Nhưng tập số thực của C là hữu hạn, hơn nữa phần lẻ của mọi số thực của C đều hữu hạn (và rất ngắn). Do đó nếu kết quả đúng cần tính ra không là số thực của C hay có phần lẻ quá dài thì ta sẽ được kết quả thực tế không còn đúng nữa. Hiện tượng này được gọi là tràn số. Bạn đã biết về tràn số nguyên trong phần Mở rộng 1.2. Còn ở đây là *tràn số thực*.

Ta có thể coi số thực của C là nỗ lực xấp xỉ số thực của Toán. Có những số có thể xấp xỉ hoàn toàn được, nghĩa là chính xác như số 1.5. Có những số không thể xấp xỉ hoàn toàn được, nghĩa là không chính xác như số 1/3 = 0.333333... hay thậm chí là số 0.1 cũng không xấp xỉ hoàn toàn được, nghĩa là không có số thực 0.1 trong C. Tại sao một số trông đơn giản như 0.1 lại không xấp xỉ hoàn toàn được? Và những câu hỏi khác như trong tràn số nguyên sẽ được trả lời một phần trong các bài sau.

BÀI TẬP

Bt 1.3.1 Viết chương trình tính chu vi và diện tích hình chữ nhật[15].

Bt 1.3.2 Viết chương trình tính chỉ số khối cơ thể (Body Mass Index):

$BMI = W/H^2$ với W là cân nặng (kg) và H là chiều cao (m).

Bt 1.3.3 Viết chương trình đổi độ F (Fahrenheit) qua độ C (Celsius):

$C = (5/9)(F - 32)$

và đổi ngược lại.

Bt 1.3.4 Nhập số thực x, tính giá trị biểu thức: $x^5 - 4x^2 + (x - 1)^2 - 1$.

---

[15] Nghĩa là chương trình cho nhập chiều dài và chiều rộng của một hình chữ nhật rồi xuất ra chu vi và diện tích của hình chữ nhật đó.

BÀI 1.4

Bạn có biết một góc vuông là bao nhiêu độ? Dĩ nhiên rồi, 90^0. Thế 2 lần góc vuông là bao nhiêu độ? Dĩ nhiên, 180^0, góc bẹt. Góc đầy, 360^0, bằng 4 lần góc vuông. Thế còn góc nhọn[1] của tam giác vuông cân? 45^0, bằng một nửa của góc vuông. Góc của tam giác đều? 60^0, bằng 2/3 góc vuông. Như vậy để "đo" góc (tức là đánh giá độ lớn của góc), người ta so tỉ lệ của nó với góc vuông. Góc 1^0 là góc bằng 1/90 góc vuông.

Một cách đo góc khác tiện lợi hơn trong Toán là so với độ dài cung. Ta đã biết đường tròn có bán kính r thì có độ dài (hay chu vi) là: $2\pi r$. Nếu bán kính là r = 1[2] thì độ dài là 2π. Thế thì độ dài cung tròn tương ứng với góc vuông là π/2 (bằng 1/4 độ dài nguyên cả đường tròn). Như vậy bằng cách dùng tỉ lệ của độ dài cung tương ứng với góc so với độ dài của cả đường tròn đơn vị ta cũng có thể đánh giá độ lớn nhỏ của góc. Góc đầy là 2π (bạn cũng có thể gọi là 2π radian), góc bẹt là π rad[3], góc vuông là π/2 rad, góc tam giác đều là π/6 rad, góc nhọn tam giác vuông cân là π/4 rad.

Câu hỏi: góc 1^0 thì là bao nhiêu radian? Trả lời: góc 1^0 là góc có độ lớn bằng 1/90 góc vuông. Mà góc vuông là π/2 rad. Vậy góc 1^0 là (π/2)/90 = π/180 rad. Câu hỏi: góc a^0 thì là bao nhiêu radian? Trả lời: góc a^0 có độ lớn bằng a lần góc 1^0. Mà góc 1^0 là π/180 rad. Vậy góc a^0 là $a \times \pi/180$ rad. Sau đây là bảng chuyển đổi độ và radian các góc đặc biệt.

Bảng 1.4.1 – Bảng chuyển đổi độ và radian các góc đặc biệt

Độ	30^0	45^0	60^0	90^0
Radian	π/6	π/4	π/3	π/2

Bạn có thể viết chương trình để in ra bảng này không?

Mã 1.4.1 – Chương trình đổi độ sang radian

```
#include <stdio.h>
int main()
{
    const double pi = 3.1416;
    printf("Do:\t%d\t%d\t%d\t%d\n", 30, 45, 60, 90);
    printf("Radian:\t%.4lf\t%.4lf\t%.4lf\t%.4lf\n",
        30*pi/180, 45*pi/180, 60*pi/180, 90*pi/180);

```

[1] Góc nhọn là góc nhỏ hơn góc vuông còn góc tù là góc lớn hơn góc vuông.
[2] Đường tròn có bán kính r = 1 còn được gọi là đường tròn đơn vị.
[3] Viết tắt của radian.


```
 9        return 0;
10    }
```

Dòng 4 giống như khai báo với khởi tạo biến nhưng có thêm từ khóa `const` đằng trước. Từ khóa này làm cho `pi` không thay đổi giá trị được nữa sau khi nó được khởi tạo. Do đó `pi` sẽ có giá trị không đổi là 3.1416. Rõ ràng `pi` không phải là biến, ta gọi nó là *hằng const*, và để tránh nhập nhằng thì các số như 3.1416 được gọi là *hằng số*. Cách dùng hằng `pi` ở trên giúp ta đỡ tốn công gõ phím. Hơn nữa, nếu ta muốn sửa giá trị của nó (chẳng hạn, xấp xỉ tốt hơn bằng 3.141569) thì ta chỉ cần sửa một lần tại chỗ khởi tạo.

Ta đã biết trong lệnh xuất thì `\n` là kí hiệu xuống dòng. Ở đây `\t` là kí hiệu tab, là kí hiệu thụt vào nhiều khoảng trắng. Ta cũng đã biết `%lf` để xuất số thực. Ở đây `%.4lf` để xuất số thực với 4 kí số sau dấu chấm thập phân[4].

Chương trình trên đã làm được nhưng chưa hay lắm. Công thức chuyển đổi đã được chép lại 4 lần. Theo nguyên lý cô đọng bạn có thể làm tốt hơn như sau.

Mã 1.4.2 – Chương trình đổi độ sang radian viết hay hơn

```
 1    #include <stdio.h>
 2
 3    double rad(double a)
 4    {
 5        return a * 3.141569 / 180;
 6    }
 7
 8    int main()
 9    {
10        printf("Do:\t%d\t%d\t%d\t%d\n", 30, 45, 60, 90);
11        printf("Radian:\t%.4lf\t%.4lf\t%.4lf\t%.4lf\n",
12            rad(30), rad(45), rad(60), rad(90));
13
14        return 0;
15    }
```

Để ý đến các Dòng 3, 4, 5, 6. Ta đã mô tả một *hàm* dùng để chuyển đổi độ sang radian, đó chính là việc "dịch qua tiếng C" của công thức chuyển đổi: góc a^0 thì là $a \times \pi/180$ rad. Bạn có thể hình dung hàm này như một cái máy chuyển đổi: bạn đưa giá trị độ vào, bạn chạy cái máy, bạn nhận kết quả là radian. Cái máy được đặt tên, gọi là *tên hàm* (giống như biến cần đặt tên và gọi là tên biến). Cần phân biệt hai thứ: một là việc mô tả cái máy, hai là việc dùng cái máy.

- Hãy xem Dòng 12 ta dùng cái máy bằng cách gọi tên máy (tên hàm), đưa *giá trị đầu vào*, và nhận *giá trị đầu ra*. Muốn có radian của 30

[4] Nếu không thì mặc định xuất ra 6 kí số sau dấu chấm thập phân.

độ hãy gọi `rad(30)`, muốn có radian của 45 độ hãy gọi `rad(45)`, …[5]. Từ ý trên, bạn có thể viết chương trình cho người dùng nhập vào độ của một góc và xuất ra radian của góc đó không?

- Dòng 3 được gọi là *prototype* của hàm hay *khai báo hàm*, nó cho biết tất cả các thông tin về cái máy cần định nghĩa: tên, máy này có bao nhiêu *dữ liệu đầu vào*[6], với từng dữ liệu đầu vào thì là tên và kiểu của dữ liệu và cuối cùng là kiểu của *dữ liệu đầu ra*[7]. Cái máy chuyển đổi của chúng ta ở trên tên là `rad`[8], cũng như tên biến, tên hàm nên gợi nhớ đến mục đích của hàm (làm gì) và cũng nên ngắn gọn. Máy này nhận một dữ liệu đầu vào là độ, đặt tên là a, kiểu số thực `double`. Máy này trả về rad là một giá trị thực nên có kiểu là `double`.
- Trong cặp ngoặc `{ … }` (các Dòng 9 đến Dòng 15) là *cài đặt* của hàm (còn gọi là *thân hàm*), đó chính là các lệnh mà hàm sẽ thực thi khi được gọi chạy, thường thì đó là công thức chuyển đổi của ta. Cũng lưu ý *từ khóa*[9] `return`.

Sau đây là khuôn mẫu khai báo và cài đặt hàm.

Hình 1.4.1 – Mẫu khai báo và cài đặt hàm

```
<kiểu dữ liệu trả về> <tên hàm>(<kiểu dữ liệu vào> <tên dữ liệu vào>)
{
    <các lệnh>
    return <biểu thức cho biết giá trị trả về của hàm>;
}
```

Bạn có tinh ý thấy rằng `main` cũng là một hàm. Để ý Dòng 8 là khai báo của hàm `main`: `int main()`. Hàm này tên `main`, không có đầu vào, trả về giá trị là số nguyên (kiểu `int`) mà cụ thể là số 0 như Dòng 14 cho thấy. Như vậy `main` cũng là hàm như `rad` (và như các hàm khác nếu có). Mà điều đó cũng có nghĩa là những gì bạn có thể làm trong `main` cũng có thể làm trong `rad` hay các hàm khác (chẳng hạn khuôn mẫu khai báo, nhập xuất). Tuy nhiên đây là hàm đặc biệt bởi cái tên đặc biệt `main`. Chạy chương trình chính là chạy hàm `main`. Mà chạy hàm `main` chính là chạy từng lệnh (theo thứ tự) trong thân hàm `main`. Mà các lệnh đó có thể gọi chạy hàm khác (như trong `main` có gọi chạy `rad`). Mà đến lượt nó, hàm được gọi chạy lại chạy từng lệnh trong thân của nó và cũng có thể gọi hàm khác nữa, …

---

[5] Bạn có thể chạy vô tư lần mà không sợ hỏng máy.

[6] Thường là 1 nhưng có thể là 0 hay nhiều.

[7] Chỉ có đúng 1 đầu ra.

[8] Bạn có thể đặt là radian hay một cái tên tùy thích, nhưng nhớ là khi bạn dùng thì dùng đúng cái tên đó.

[9] Ta sẽ biết về từ khóa trong bài sau.

Tinh ý hơn nữa bạn có thể thấy các lệnh nhập xuất chẳng qua là lệnh gọi chạy các hàm tương ứng là `printf`, `scanf`, `gets`. Chẳng hạn Dòng 10 là gọi chạy hàm `printf`. Khác với hàm `rad` của ta chỉ có một đầu vào thì `printf` có thể có một hoặc nhiều đầu vào (và nếu nhiều thì dùng dấu `,` để phân cách các đầu vào). Cũng lưu ý là không có lời gọi chạy hàm `main` vì hàm `main` được *môi trường thực thi*[10] gọi chạy tự động khi chạy chương trình. Câu hỏi là hàm `printf` được định nghĩa ở đâu mà không thấy? Trả lời: `printf` được định nghĩa sẵn trong `stdio.h`[11]. Đó là lí do mà có Dòng 1: `#include <stdio.h>`[12]. Nếu không có chỉ thị này (thử đi nhé) bạn sẽ thấy báo lỗi đại khái là không có định nghĩa của `printf`. Hiển nhiên là nếu không có định nghĩa của một hàm thì ta không thể gọi thực thi hàm đó được (vì đâu biết thực thi thế nào). Những hàm như `printf`, `scanf`, `gets` là những hàm đã được *định nghĩa sẵn*[13], ta có thể dùng mà không cần tự mình định nghĩa lấy.

Trở lại với góc, bạn có biết sin, cos của góc không. Nghe như đâu đó "tìm sin lấy đối chia huyền, cosin lấy cạnh kề, huyền chia nhau". Nếu được cho một tam giác vuông, ta có thể tìm sin, cos các góc nhọn của nó bằng cách đo các cạnh của tam giác vuông và thực hiện phép chia như câu thần chú trên. Chẳng hạn, $\sin 45^0$ hay $\cos 45^0$ đều là $\sqrt{2}/2$, $\sin 30^0 = 1/2 = \cos 60^0$, $\sin 60^0 = \sqrt{3}/2 = \cos 30^0$. Sau đây là bảng sin, cos các góc đặc biệt.

Bảng 1.4.2 – Bảng sin, cos các góc đặc biệt

Độ	30^0	45^0	60^0	90^0
Radian	π/6	π/4	π/3	π/2
Sin	1/2	$\sqrt{2}/2$	$\sqrt{3}/2$	1
Cos	$\sqrt{3}/2$	$\sqrt{2}/2$	1/2	0

Thế còn $\sin 20^0$ hay $\cos 10^0$ là bao nhiêu? Ta không thể dùng câu thần chú trên vì không có tam giác vuông đặc biệt nào để dùng. Bạn có thể viết hàm (tức là định nghĩa) sin, cos không? Quá khó rồi![14]. Ta thậm chí không biết công thức hay cách làm nên không thể viết hàm được[15]. Rất may, sin, cos cũng là những hàm được định nghĩa sẵn (với tên hàm là `sin`, `cos`). Không

[10] Chính xác môi trường thực thi là gì thì ta sẽ tìm hiểu sau, hiện giờ ta có thể hiểu đó là *hệ điều hành*, người quản lý các chương trình.
[11] Thật ra chỉ có prototype trong đó mà không có thân hàm, câu chuyện bí ẩn này sẽ được tìm hiểu sau.
[12] Cái này được gọi là *chỉ thị* `#include`. Chỉ thị là gì thì ta sẽ tìm hiểu sau, tạm thời ta chỉ cần biết nó không phải là lệnh.
[13] Những hàm được định nghĩa sẵn thường là những hàm quan trọng, hay dùng.
[14] Bạn sẽ biết cách tự mình định nghĩa những hàm như thế này khi công lực cao hơn.
[15] Ta viết được hàm đổi độ sang radian vì ta đã biết công thức đổi.

chỉ vậy, C cũng định nghĩa sẵn hàm tính căn (với tên hàm là `sqrt`[16]) và các hàm tính toán thông thường khác như hàm lũy thừa, mũ, logarit, … Khác với `printf`, `scanf` các hàm Toán này được khai báo trong `math.h`.

Tập các hàm có sẵn được gọi là *thư viện chuẩn* của C. Chúng được gom lại trong các file `.h` theo mục đích sử dụng:

- Các *hàm nhập xuất* trong `stdio.h`[17] như `printf`, `scanf`, `gets`, …
- Các *hàm tính toán* trong `math.h`[18] như `sin`, `cos`, `sqrt` (tính căn bậc hai), `pow` (tính x^y), `exp` (tính e^x), `log` (tính ln$x$), …
- và các hàm khác trong các file `.h` khác.

Để dùng những hàm có sẵn này bạn cần biết chúng nằm trong file `.h` nào và thêm chỉ thị tương ứng ở đầu chương trình như ta đã làm với hàm `printf` hay `scanf` trong `stdio.h`:

```
#include <stdio.h>
```

Lẽ dĩ nhiên, những hàm không có sẵn thì ta phải tự định nghĩa lấy; do đó, chúng được gọi là *hàm do lập trình viên tự định nghĩa*. Kiểu gì đi nữa, để thực thi một hàm thì hàm đó phải được định nghĩa rồi: nếu không có chỉ thị `#include <stdio.h>` khi dùng hàm `printf` thì C sẽ báo lỗi "`printf` không được định nghĩa". Cũng vậy, nếu bỏ định nghĩa của `rad` hay dời nó xuống sau hàm `main` thì C cũng báo lỗi "`rad` không được định nghĩa". Tóm lại: hàm phải được định nghĩa trước khi dùng cũng như biến phải được khai báo trước khi dùng.

Một nguyên lý cần khắc cốt ghi tâm: "*tận dụng tối đa những thứ có sẵn*". Nếu bạn cần làm gì đó mà đã có sẵn hàm thì hãy dùng nó, nếu không thì bạn phải định nghĩa nhưng hãy tận dụng tối đa những hàm đã có. Nếu bạn cần cái bánh xe thì hãy dùng cái bánh xe vì nó đã có sẵn ("không phát minh lại cái bánh xe"), còn như cần cái xe đạp thì hãy tận dụng 2 cái bánh xe có sẵn (và các thứ có sẵn khác nữa để ráp thành cái xe đạp).

Cuối cùng, nhiệm vụ quan trọng cho bạn đây: hãy mở rộng chương trình trên để xuất ra bảng đầy đủ có sin, cos các góc đặc biệt 0^0, 30^0, 45^0, 60^0, 90^0. Lưu ý: bạn có thể tính sin, cos góc bất kì, miễn là đưa cho nó radian của góc. Chẳng hạn góc 45^0 thì là $\pi/4$ rad nên để tính sin45^0 bạn gọi `sin(pi/4)`. Với các góc đặc biệt, bạn còn có thể tính sin, cos bằng các phép chia như bảng trên. Chẳng hạn, để tính sin45^0 bạn gọi `sqrt(2)/2`. Hãy làm theo cả hai cách để đối chiếu kết quả.

MỞ RỘNG 1.4 – Tính giá trị đa thức bằng phương pháp Horner

---

[16] Viết tắt của square root.

[17] stdio là viết tắt của standard input output.

[18] math là viết tắt của mathematics.

Bạn có tính được giá trị của biểu thức $P(x) = x^4 + 4x^3 + 6x^2 + 5x + 2$ khi x nhận một giá trị số được cho nào đó không? Chẳng hạn, với $x = 2$ thì $P(x)$ có giá trị bao nhiêu? Bằng cách thay x bằng 2 và thực hiện các phép toán cộng (trừ) và nhân ta sẽ có giá trị của $P(2)$[19]. Thế còn với $x = 9.9$ thì $P(x)$ có giá trị bao nhiêu? Nếu kiên nhẫn không phải là đức tính của bạn thì bạn có thể viết chương trình C như sau để trả lời.

Mã 1.4.3 – Chương trình tính giá trị đa thức

```
#include <stdio.h>
int main()
{
    double x = 9.9;
    double P = x*x*x*x + 4*x*x*x + 6*x*x + 5*x + 2;
    printf("P(%lf) = %lf\n", x, P);

    return 0;
}
```

Chỉ cần sửa giá trị tương ứng của x, chương trình trên giúp ta tính giá trị của $P(x)$ với mọi giá trị được cho của x. Bạn cũng có thể sửa chương trình trên để nó tính giá trị của $P(x)$ với giá trị người dùng nhập cho x.

Tuy nhiên điều tôi muốn đố bạn là: có cách nào tính $P(x)$ nhanh hơn không? Để tính $P(x)$ thì chương trình trên dùng 9 phép nhân và 4 phép cộng[20]. Số lượng phép toán này có thể nói là không đáng kể với máy[21]. Tuy nhiên, ta có thể tính nhanh hơn bằng nhận xét sau:

$$\begin{aligned} P(x) &= x^4 + 4x^3 + 6x^2 + 5x + 2 \\ &= (x^3 + 4x^2 + 6x + 5)x + 2 \\ &= ((x^2 + 4x + 6)x + 5)x + 2 \\ &= (((x + 4)x + 6)x + 5)x + 2 \end{aligned}$$

Với dòng cuối cùng, ta có thể tính $P(x)$ chỉ với 3 phép nhân và 4 phép cộng. Wow! Cách tính này mà ta gọi là *tính kiểu Horner*[22] đã giúp giảm từ 9 phép nhân chỉ còn 3 phép nhân (số lượng phép cộng thì như nhau). Nếu bạn không ấn tượng về điều này thì cũng bình thường. Bạn hãy viết một chương trình nữa tính $P(x)$ theo cách mới này và so thời gian chạy của hai chương trình. Kết quả: cả hai đều tính xong $P(x)$ trong tích tắc. 9 phép nhân hay 3 phép nhân thì với máy đều không đáng kể. Thực sự thì chương trình mới chạy nhanh hơn nhưng bạn cần có đặc dị công năng để quan sát thấy:)

[19] Đáp án: 84.
[20] Đếm lại giùm tôi xem thử đúng không nhé.
[21] Mặc dù có thể là cả vấn đề với ta.
[22] Đặt theo tên nhà toán học Anh William George Horner.

Tổng quát hơn, biểu thức như *P*(*x*) ở trên được gọi là *đa thức*, số mũ cao nhất của đa thức được gọi là *bậc của đa thức*, chẳng hạn, *P*(*x*) là đa thức bậc 4. Bằng những phân tích chi tiết[23], người ta thấy rằng: một đa thức bậc *N*, nếu tính theo cách đầu (mà ta còn gọi là cách tính ngây thơ) sẽ cần khoảng N^2 phép nhân, còn tính theo cách sau (cách tính Horner) sẽ cần khoảng *N* phép nhân, cả hai đều cần khoảng *N* phép cộng. Như vậy để tính đa thức có bậc cỡ 1 triệu, giả sử mỗi phép nhân cần tốn 1 µs (10^{-6} giây) thì cách tính Horner chỉ tốn 1 giây nhưng cách tính ngây thơ tốn 1 triệu giây (là khoảng 12 ngày). Khác biệt là quá rõ rồi nhé!

Thêm một điều nữa: về mặt phương pháp hay cách thức giải quyết vấn đề, Horner đã đưa ra một phương pháp tính rất hay, rất đáng học hỏi để giải quyết những vấn đề hay bài toán khác. Mặc dù bí kíp này không nhằm giúp bạn đưa ra được những phương pháp như vậy, nhưng ta cũng sẽ gặp nhiều ví dụ và nên học cách vận dụng chúng.

BÀI TẬP

Bt 1.4.1 Viết chương trình tính diện tích tam giác biết chiều dài 3 cạnh tam giác là *a*, *b*, *c* (giả sử đó là các chiều dài hợp lệ).

<u>*Gợi ý*</u>: Dùng công thức Heron

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

với *p* là nửa chu vi, $p = (a + b + c)/2$.

Bt 1.4.2 Dùng các hàm toán trong `math.h` viết chương trình cho nhập số thực *x*, tính và xuất ra các giá trị *y*, *z* lấy 2 chữ số lẻ:

a) $y = \left(\sqrt[3]{x} + 10x\sqrt{|x|} + \frac{3}{x^{-5}}\right)^{0.1}$

b) $z = \frac{\sin \pi x^2 + \sqrt[4]{x^4+1} + \ln 10}{e^{2x} + 10^{\cos\left(\frac{\pi}{4}+x\right)} + \log_2 0.1}$

Bt 1.4.3 Viết chương trình cho người dùng nhập số thực dương y, đặt $x_0 = \frac{y}{2}, x_1 = \frac{x_0}{2} + \frac{y}{2x_0}, x_2 = \frac{x_1}{2} + \frac{y}{2x_1}, \ldots, x_5 = \frac{x_4}{2} + \frac{y}{2x_4}$. Xuất ra x_5 và $\sqrt{y}$. Nhận xét.

Bt 1.4.4 Viết các hàm và chương trình dùng các hàm này để có chương trình như Bài tập tương ứng. (*Xem Hướng dẫn giải bên dưới*)

a) Nhận tên người rồi xuất ra "Bài Happy birthday" cho người đó. (Bài tập 1.2.1)

b) Tính tổng các chữ số của một số nguyên dương có 3 chữ số. (Bài tập 1.2.4)

c) Đổi độ *F* qua độ *C* theo công thức *C* = (5/9)(*F* – 32). (Bài tập 1.3.3)

---

[23] Mà ta cũng sẽ học cách làm như vậy khi công lực tăng cao.

d) Tính diện tích tam giác từ chiều dài 3 cạnh tam giác (giả sử đó là các chiều dài hợp lệ). (Bài tập 1.4.1)

Bt 1.4.5 Viết chương trình tính giá trị đa thức $x^5 - 4x^3 + (x - 1)^2 - 2$ bằng phương pháp Horner với *x* nhận giá trị từ người dùng.

Hướng dẫn giải Bài tập 1.4.4

Yêu cầu: Viết hàm tính chỉ số khối cơ thể từ cân nặng (kg), chiều cao (m) và chương trình dùng hàm này để có chương trình như Bài tập 1.3.2.

Giải: Để viết một hàm trước hết ta cần xác định công việc của hàm đó (hàm đó làm gì?). Sau đó xác định các dữ liệu đầu vào và dữ liệu đầu ra của hàm. Từ đó viết prototype của hàm (bao gồm việc chọn tên phù hợp) và cài đặt công việc của hàm trong thân hàm. Chẳng hạn công việc của hàm ta cần viết là tính chỉ số khối cơ thể (BMI) từ cân nặng (tính theo kg) và chiều cao (tính theo m). Hàm này có 2 dữ liệu đầu vào là cân nặng và chiều cao. Đây đều là các số thực (do có thể lẻ như nặng 45.5 kg hay cao 1.65 m). Dữ liệu đầu ra là chỉ số BMI, là một số thực. Chọn tên phù hợp cho hàm là `BMI` (nghĩa là tính BMI), tên cho dữ liệu đầu vào cân nặng là `weight` (nghĩa là cân nặng), tên cho dữ liệu đầu vào chiều cao là `height` (nghĩa là chiều cao). Kiểu của 2 dữ liệu đầu vào và dữ liệu đầu ra đều là `double`. Thân hàm sẽ cài đặt công thức tính: BMI = $(\textit{Cân nặng})/(\textit{Chiều cao})^2$. Để viết chương trình, ta viết hàm `main` khai báo biến phù hợp, cho nhập dữ liệu, gọi hàm `BMI` và xuất ra dữ liệu tính toán từ giá trị trả về của hàm (nều cần). Tóm lại ta có thể viết như sau.

Mã 1.4.4 – Chương trình đổi độ sang radian viết hay hơn

```c
#include <stdio.h>
double BMI(double weight, double height)
{
    return weight / (height*height);
}
int main()
{
    double w, h;

    printf("Can nang (kg): ");
    scanf("%lf", &w);
    printf("Chieu cao (m): ");
    scanf("%lf", &h);

    printf("Chi so BMI = %.2lf", BMI(w, h));

    return 0;
}
```

BÀI 1.5

Một đa thức bậc hai[1] là một biểu thức có dạng: $ax^2 + bx + c$. Trong đó a, b, c được gọi là các hệ số và là các số thực được cho với $a \neq 0$, x là một kí hiệu hình thức gọi là biến. Biểu thức này sẽ có giá trị (là một số thực) khi ta thay một số thực nào đó vào x và tính biểu thức. Chẳng hạn đa thức bậc hai: $x^2 + 2x + 1$ có các hệ số là $a = 1$, $b = 2$, $c = 1$. Khi thay $x = 1$ thì đa thức này có giá trị là 4, chính là kết quả của biểu thức $1^2 + 2\times1 + 1$. Nếu một số thực x_0 nào đó khi thay vào x làm cho đa thức có giá trị là 0 thì x_0 được gọi là nghiệm của đa thức. Chẳng hạn nếu thay -1 vào đa thức trên thì giá trị của đa thức là 0 (là giá trị của biểu thức $(-1)^2 + 2\times(-1) + 1$), do đó -1 là nghiệm của đa thức trên. Việc tìm nghiệm của đa thức là tìm ra tất cả các nghiệm của đa thức đó. Đa thức trên ngoài nghiệm -1 thì còn có nghiệm nào khác nữa không? Người ta chứng minh được là đa thức trên chỉ có một nghiệm là -1. Bài toán tìm nghiệm của đa thức là bài toán quan trọng, có nhiều ứng dụng trong thực tế. Chẳng hạn bài toán: tìm hai số biết tổng của chúng là 5 và tích của chúng là 6 chính là bài toán tìm nghiệm của đa thức bậc hai: $x^2 - 5x + 6$.[2]

Làm sao để tìm nghiệm của một đa thức bậc hai? Từ định nghĩa của nghiệm, ta có ngay cách tìm nghiệm gọi là "*mò nghiệm*"[3]: thử thay số nào đó vào x và kiểm tra nếu giá trị của biểu thức là 0 thì đó là nghiệm còn khác 0 thì không là nghiệm. Cách làm này rất đơn giản nhưng tiếc là không dùng được vì ta có vô số số thực có khả năng là nghiệm.

Rất may, người ta đã chứng minh được rằng một đa thức bậc hai có thể không có nghiệm, có một nghiệm hoặc có hai nghiệm và nghĩ ra cách tìm nghiệm hiệu quả như sau:

- Với đa thức bậc hai: $ax^2 + bx + c$ ($a \neq 0$).
- Tính $\Delta = b^2 - 4ac$.
- Nếu $\Delta < 0$ thì đa thức vô nghiệm.
- Nếu $\Delta = 0$ thì đa thức có một nghiệm $x = \frac{-b}{2a}$.
- Nếu $\Delta > 0$ thì đa thức có hai nghiệm $x_1 = \frac{-b+\sqrt{\Delta}}{2a}, x_2 = \frac{-b-\sqrt{\Delta}}{2a}$.

Bạn đừng hỏi rằng tại sao lại làm như vậy hay làm như vậy có đúng không. Ở đây ta không bàn về chuyện đó. Đó là câu chuyện khác. Việc của ta

[1] Còn gọi là tam thức.

[2] Tại sao? Bạn có thể coi lại SGK Toán (lớp 9 hay 10 gì đó), còn không thì bạn có thể thế vào để kiểm tra nghiệm của đa thức trên là 2, 3 và đó cũng là hai số cần tìm.

[3] Lịch sự hơn thì gọi là *dò nghiệm*:)

sẽ là viết chương trình C từ cách làm được cho ở trên để tìm nghiệm của đa thức bậc hai. Bản mô tả cách làm trên còn được gọi là *mã giả*. Đó là công việc (hay cách làm) được mô tả bằng ngôn ngữ tự nhiên ngắn gọn, thường pha lẫn các kí hiệu toán. Như vậy việc của ta là dịch mã giả trên qua mã thật - mã C - chương trình C, là công việc (hay cách làm) được mô tả bằng ngôn ngữ C. Sau đây là một bản dịch tốt.

Mã 1.5.1 – Chương trình tìm nghiệm đa thức bậc hai

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      double a, b, c, delta;
6
7      printf("Nhap cac he so: ");
8      scanf("%lf %lf %lf", &a, &b, &c);
9
10     delta = b*b - 4*a*c;
11     if(delta < 0)
12         printf("Da thuc vo nghiem.\n");
13     if(delta == 0)
14         printf("Da thuc co 1 nghiem: %lf.\n", -b/(2*a));
15     if(delta > 0)
16     {
17         double x1 = (-b - sqrt(delta))/(2*a);
18         double x2 = (-b + sqrt(delta))/(2*a);
19         printf("Da thuc co 2 nghiem: %lf, %lf.\n",
20                      x1, x2);
21     }
22
23     return 0;
24 }
```

Hãy học hỏi cách nhập nhanh 3 số thực ở Dòng 8[4]. Khi chạy chương trình, ta gõ 3 số cách nhau bằng khoảng trắng rồi nhấn Enter. Kết quả chạy với trường hợp người dùng muốn tìm nghiệm của đa thức $x^2 - 5x + 6$.

```
Nhap cac he so: 1 -5 6↵
Da thuc co 2 nghiem: 2.000000, 3.000000.
```

Điểm mới của chương trình trên chính là khuôn mẫu "*Nếu - thì*", chi tiết hơn là "Nếu *A* đúng thì làm *B*":

`if`(<điều kiện *A*>)
    <công việc *B*>

[4] Hi vọng là bạn thấy được khuôn mẫu.

Trong đó *A* là một điều kiện nào đó còn *B* là công việc cần làm khi điều kiện *A* đúng. Cụ thể thì *A* là một *biểu thức luận lý*, là biểu thức mà giá trị là *đúng* hoặc *sai*. Biểu thức luận lý thông thường nhất là *biểu thức so sánh* các số, với các toán tử > (lớn hơn), >= (lớn hơn hoặc bằng), < (nhỏ hơn), <= (nhỏ hơn hoặc bằng), == (bằng, lưu ý là hai dấu = viết liền và phân biệt với toán tử = của phép gán) và != (không bằng). Công việc *B* có thể là một lệnh đơn giản nào đó như lệnh xuất ở Dòng 12, hoặc có thể là một *khối* như ở các Dòng 16 đến Dòng 21. Một khối mô tả công việc phức tạp và có khuôn mẫu:

```
{
        <Các công việc>
}
```

Bạn đã gặp khối này ở đâu? Ở thân hàm (mà thân hàm `main` là một trường hợp). Như vậy tất cả những gì các bạn có thể làm trong hàm `main` hay các hàm khác đều có thể làm trong khối (như khai báo, nhập, xuất, tính toán, gán). Khối ở các Dòng 16 đến Dòng 21 gồm hai khai báo với khởi tạo và một lệnh xuất. Bạn có thể viết gọn lại bằng một lệnh xuất như sau:

```
if(delta > 0)
    printf("Da thuc co 2 nghiem: %lf, %lf.\n",
        (-b - sqrt(delta))/(2*a),
        (-b + sqrt(delta))/(2*a));
```

và khi đó không cần dùng khối vì công việc chỉ là một lệnh đơn. Gọn nhưng rối! Theo tiêu chí rõ ràng thì viết như cũ tốt hơn.

Một cách tự nhiên, bạn cũng có thể đoán là có khuôn mẫu "Nếu *A* đúng thì làm *B* còn không thì làm *C*" như sau:

```
if(<điều kiện A>)
    <công việc B>
else
    <công việc C>
```

Khuôn mẫu "Nếu - thì" trên cho phép ta lựa chọn có làm việc gì đó hay không dựa trên điều kiện nào đó. Nó, như vậy, được gọi là một *cấu trúc lựa chọn*. Ta cũng thường xuyên cần làm lặp lại một công việc nào đó. Đó là các *cấu trúc lặp* mà ta sẽ tìm hiểu vài khuôn mẫu sau đây.

Giả sử Valentine sắp tới, bạn muốn thể hiện tình cảm của mình với người ấy. Thay vì tặng 1000 đóa hồng rất tốn kém và lãng phí, bạn tặng một chương trình C xuất ra nhiều lần thông điệp thầm kín "I love you!":) Chẳng hạn như chương trình sau.

Mã 1.5.2 – Chương trình Valentine[5]

```
#include <stdio.h>
int main()
{
    printf("I love you!\n");
    printf("I love you!\n");
    printf("I love you!\n");
    printf("I love you!\n");
    printf("I love you!\n");

    return 0;
}
```

Tưởng tượng rằng mỗi dòng "I love you!" là một bông hồng thì bạn đã tặng người ấy tới 5 bông hồng. Và nếu như 5 là hơi ít thì bạn có thể tặng người ấy nhiều hơn, 100 chẳng hạn, bằng cách gõ 100 lần lệnh xuất như vậy. Cũng hơi tốn công (copy-paste sẽ giúp bạn phần nào) nhưng đỡ tốn kém:) Nhưng nếu như bạn muốn tăng mức độ, lên 1000 hay thậm chí gần như vô hạn, 1 triệu đóa hồng thì sao? Tôi không chắc là bạn đủ kiên nhẫn để copy-paste đâu. Hãy làm như sau.

Mã 1.5.3 – Chương trình Valentine ít tốn công:)

```
#include <stdio.h>
int main()
{
    for(int i = 1; i <= 1000; i = i + 1)
        printf("I love you!\n");

    return 0;
}
```

Vâng: 1000 đóa hồng không tốn kém cũng chẳng tốn công:) Bạn biết sửa chỗ nào để tặng người ấy 1 triệu hay thậm chí 1 tỉ bông hồng rồi đó. Bạn hãy tự tìm ra khuôn mẫu "*Làm N lần*" từ chương trình trên.

Ta có thể dùng khuôn mẫu trên để làm công việc hữu ích hơn sau đây:) Tính giai thừa. Cho một số nguyên dương n, *giai thừa* của n, kí hiệu $n!$, được định nghĩa là: $n! = 1\times2\times3\times \ldots \times n$. Chẳng hạn giai thừa của 4 là 24 ($4! = 1\times2\times3\times4 = 24$). Vì hàm tính giai thừa chưa có trong thư viện chuẩn của C nên ta sẽ viết hàm này như sau.

Mã 1.5.4 – Hàm tính giai thừa

```
int fact(int n)
{
```

[5] Nếu như "I love you!" hơi lộ liễu thì bạn có thể thử dùng `printf("I \x03 you!\n")`.

```
3       int f = 1;
4       for(int i = 1; i <= n; i = i + 1)
5           f = f * i;
6       return f;
7   }
```

Ta cũng dùng khuôn mẫu "Làm *N* lần công việc *A*" nhưng lần này ta có dùng *biến lặp* (biến `i` ở trên) trong công việc A (xem biến lặp `i` trong lệnh gán ở Dòng 5). Bạn hãy viết chương trình cho người dùng nhập vào một số nguyên dương rồi xuất ra giai thừa của nó bằng cách dùng hàm tính giai thừa trên.

Một cấu trúc lặp nữa cũng hay dùng là khuôn mẫu "*Làm - trong khi*". Ta sẽ viết chương trình cho người dùng nhập vào các số nguyên và tính tổng các số này trong khi người dùng không nhập số 0 (nghĩa là *cho đến khi* người dùng nhập số 0 thì dừng) và xuất ra tổng các số đã nhập.

Mã 1.5.5 – Chương trình tính tổng các số người dùng nhập

```
#include <stdio.h>
int main()
{
    int x, s = 0;
    printf("Nhap cac so nguyen:\n");
    do
    {
        scanf("%d", &x);
        s = s + x;
    }
    while(x != 0);
    printf("Tong cac so da nhap: %d\n", s);

    return 0;
}
```

Hãy dọc thử để đảm bảo là bạn thấy rõ khuôn mẫu đó nhé.

MỞ RỘNG 1.5 – Tính căn bằng phương pháp chia đôi

Làm sao để tính căn bậc hai của một số thực dương? Quá dễ! Trong C hãy dùng hàm `sqrt` được định nghĩa sẵn trong `math.h`. Bạn có thể tự mình định nghĩa hàm này không? Quá khó! Trong khi phép toán ngược lại của căn là phép bình phương thì quá dễ để làm. Cho x để tính $y = x^2$, ta chỉ cần nhân x với x (trong C là `x*x`). Ngược lại, cho y (dương) làm sao để tính $x_0 = \sqrt{y}$? x_0 chính là nghiệm dương của phương trình $x^2 = y$ (với y là số thực dương được cho). Chẳng hạn $\sqrt{2}$ chính là nghiệm dương của phương trình $x^2 = 2$. Vậy thì giải phương trình bậc hai này ta có thể tìm được nghiệm x_0. Nghiệm của phương trình bậc hai $x^2 = y$ chính là nghiệm

của đa thức bậc hai: $x^2 + 0x - y$ (x là biến, y là hệ số). Vậy thì dùng cách giải đa thức bậc hai ở trên ta có thể tìm được nghiệm, tức là tính được căn rồi. Rất tiếc cách giải ở trên đòi hỏi phải dùng phép toán tính căn, chính là phép toán ta đang muốn định nghĩa. Vấn đề là ta muốn tính căn bằng các *phép toán cơ bản* là cộng, trừ, nhân, chia. Vậy thì ta cũng giải phương trình $x^2 = y$ nhưng bằng phương pháp mà ta đã chê – *mò nghiệm*: thử các giá trị x khác nhau để tìm ra giá trị mà x^2 bằng y dương được cho. Phương pháp này về cơ bản là không khả thi do có vô số khả năng của x mà ta phải thử. Tuy nhiên ta có hai nhận xét quan trọng sau đây:

- (1): x_0 sẽ nằm trong khoảng từ 0 đến y nếu $y > 1$ và từ 0 đến 1 nếu $y \leq 1$.
- (2): x^2 là hàm đồng biến khi x dương, nghĩa là với $0 < x_1 \leq x_2$ thì $x_1^2 \leq x_2^2$.

Từ hai nhận xét trên ta có thể dùng một phương pháp mò nghiệm gọi là *phương pháp chia đôi*. Cụ thể ta sẽ mò nghiệm trong khoảng [l, r] (với $l \leq r$) và cố thu hẹp khoảng này cho đến khi tìm thấy nghiệm. Ban đầu, do nhận xét (1) ta có $l = 0$ và $r = \max(y, 1)$. Bằng cách thử giá trị ở giữa khoảng: $x = (l + r)/2$. Nếu $x^2 = y$ thì ta đã tìm thấy nghiệm $x_0 = x$, còn nếu $x^2 > y$ thì theo nhận xét (2) ta có nghiệm x_0 sẽ nằm trong khoảng [l, x], còn nếu $x^2 < y$ thì cũng theo nhận xét (2) ta có nghiệm x_0 sẽ nằm trong khoảng [x, r]. Bằng cách lặp lại việc này ta sẽ thu hẹp khoảng chứa nghiệm x_0 cho đến khi tìm thấy. Mỗi lần lặp thì khoảng chứa nghiệm sẽ được thu hẹp đi một nửa nên phương pháp này có tên gọi là chia đôi.

Phương pháp chia đôi rất đơn giản và có thể dùng để tìm nghiệm của nhiều loại phương trình (và giải được nhiều bài toán) khác nhau. Nếu ta tự mình thực hiện các phép toán thì phương pháp này không khả thi do số lần lặp có thể rất lớn, nhưng nếu ta viết chương trình để máy tính toán thì phương pháp này là hoàn toàn khả thi và là phương pháp hay do tính đơn giản và khả năng ứng dụng lớn của nó.

Còn một điều nữa trước khi ta bắt tay viết chương trình. Các số thực trên máy chỉ là xấp xỉ của các số thực trong Toán (xem Mở rộng 1.3) cho nên việc kiểm tra đẳng thức $x^2 = y$ (trong C là biểu thức so sánh `x*x == y`) có thể không còn chính xác do cả x lẫn y trên máy đều là các giá trị xấp xỉ. Nói chung là nên tránh kiểm tra bằng trên các số thực. Để kiểm tra xem đã tìm thấy nghiệm x_0 trong khoảng [l, r] hay chưa ta sẽ kiểm tra xem khoảng này đủ hẹp hay chưa, nếu sai số tương đối $(r - l)/r \leq \varepsilon$ (với ε là một số thực dương rất nhỏ chọn trước) thì nghĩa là đã tìm thấy. Theo khuôn mẫu "Làm – trong khi" và "Nếu - thì", ta có tự mình viết hàm tính căn như Mã 1.5.6. Ở đây ta chọn $\varepsilon = 0.0000001$. Cũng lưu ý là ε không được nhỏ hơn sai số tương đối nhỏ nhất trên máy gọi là *ε của máy* (machine

epsilon). Với kiểu double của C thì giá trị đó là khoảng 10^{-16}.

Mã 1.5.6 – Hàm tính căn bằng phương pháp chia đôi

```
double mysqrt(double y)
{
    double x, l, r;

    if(y > 1)
        r = y;
    else
        r = 1;
    l = 0;
    do
    {
        x = (l + r)/2;
        if(x * x > y)
            r = x;
        else
            l = x;
    }
    while ((r - l)/r > 0.0000001);

    return x;
}
```

Bạn hãy viết chương trình cho người dùng nhập một số thực dương rồi dùng hàm mysqrt trên để tính căn và so kết quả với hàm sqrt trong thư viện chuẩn. Bạn cũng thử dùng phương pháp chia đôi trên để tính căn bậc 3. Hơn nữa, hãy nhớ phương pháp này, nó sẽ rất hữu ích về sau.

BÀI TẬP

Bt 1.5.1 Tìm số lớn nhất, nhỏ nhất trong 2 số.

Bt 1.5.2 Tìm số lớn nhất, nhỏ nhất trong 3 số.

Bt 1.5.3 Tính chỉ số BMI (Bài tập 1.3.2) và phân loại theo bảng sau đây cho người lớn[6]:

- dưới 18.5: gầy.
- từ 18.5 đến dưới 25: bình thường.
- từ 25 đến dưới 30: béo.
- từ 30 trở lên: béo phì.

Bt 1.5.4 Giải phương trình: $ax + b = 0$. (Nghĩa là tìm nghiệm của đa thức bậc nhất: $ax + b$ với a, b là các hệ số thực.)

[6] Theo WHO – BMI classification.

Bt 1.5.5 Tính các tổng sau bằng cách lặp (dùng các cấu trúc lặp):

- $S_1(n) = 1 + 2 + \ldots + n$
- $S_2(n) = 1^2 + 2^2 + \ldots + n^2$

và kiểm tra lại các đẳng thức:

- $S_1(n) = n(n + 1)/2$
- $S_2(n) = n(n + 1)(2n + 1)/6$

Bt 1.5.6 Đếm số chữ số của một số nguyên không âm (không kể các chữ số 0 ở đầu).

Bt 1.5.7 Tính tổng các chữ số của một số nguyên không âm.

Bt 1.5.8 Nhập một số nguyên và xuất ra số nguyên đó với các dấu phẩy (,) phân cách mỗi 3 chữ số. Ví dụ: nhập 1234567 xuất ra `1,234,567`.

Bt 1.5.9 Từ ý tưởng của Bài tập 1.4.3, hãy viết chương trình tính gần đúng căn bậc 2 của một số thực dương[7].

Bt 1.5.10 Dùng phương pháp chia đôi trong phần mở rộng để tính căn bậc 3 của một số thực dương.

---

[7] Phương pháp tính căn này được gọi là phương pháp Newton.

BÀI 1.6

Tôi chắc một điều là bạn đã không thể chạy suôn sẻ tất cả các chương trình từ bài đầu đến giờ. Rất dễ xảy ra *lỗi* khi viết chương trình, nhất là với những người mới bắt đầu. Cũng như khi bạn mới tập làm văn, bài văn bạn viết ra thường có rất nhiều lỗi. Khác với các bài văn, một số lỗi nhỏ có thể được du di bỏ qua, một lỗi (dù chỉ một lỗi nhỏ nhất) cũng làm cho chương trình C của bạn không chạy được hay để lại hậu quả ghê gớm![1]. Đó là lý do mà bài học này sớm xuất hiện với mục đích là không để bạn bỏ cuộc quá sớm:) Bạn cũng sẽ thấy rằng việc mắc lỗi trong chương trình C cũng tự nhiên như cách bạn hay mắc lỗi trong cuộc sống vậy!:). Khi công lực tăng cao, bạn sẽ viết chương trình có ít lỗi hơn. Nhưng, có thể nói rằng, mọi chương trình đều có lỗi hoặc tìm ẩn lỗi.

Bạn hãy gõ và chạy chương trình sau.

Mã 1.6.1 – Chương trình Hello đầy lỗi:)

```
#include <stdio.h>
int Main()
{
    char name[1oo];
    printf("What's your name? );
    scanf("%s", &name)
    printf("Hello %s", name);

    return 0;
}
```

Sorry, chương trình này không thể chạy được!:) Đây là "chương trình Hello đầy lỗi". Ta sẽ tìm và sửa tất cả các lỗi để có chương trình tốt. Có bao nhiêu lỗi trong chương trình này? Bạn sẽ ngạc nhiên về con số.

Trước hết, phải thú nhận là tôi đã phỉnh bạn rằng ta phải viết chương trình bằng ngôn ngữ C (hay một ngôn ngữ lập trình nào đó) vì máy chỉ hiểu ngôn ngữ lập trình mà không hiểu ngôn ngữ tự nhiên như tiếng Việt, tiếng Anh. Thật ra, cả ngôn ngữ C, máy cũng không hiểu được. Máy chỉ hiểu ngôn ngữ của nó: *ngôn ngữ máy*. Ngôn ngữ máy (và ngôn ngữ rất gần gũi với nó, *hợp ngữ*[2]) có nhược điểm là quá chi tiết và thay đổi theo từng loại/hệ máy, hơn nữa nó cũng không hợp với ta (là con người) nên ta không muốn dùng

[1] Có lẽ nó giống các văn bản luật.

[2] Ngôn ngữ máy và hợp ngữ không phải là đối tượng của tài liệu này nhưng cũng sẽ được tìm hiểu phần nào lúc thích hợp.

nó để viết chương trình[3]. Do đó, ta viết chương trình bằng ngôn ngữ C, và chương trình này cần được *biên dịch* ra ngôn ngữ máy để máy có thể thực thi. Điều này cũng tương tự như một cuốn sách tiếng Anh cần được biên dịch thành tiếng Việt để những người Việt (không biết hoặc không muốn biết tiếng Anh) có thể đọc hiểu được.

Vậy thì ai hay cái gì chịu trách nhiệm làm công việc biên dịch này? Một hệ thống trong công cụ mà bạn đã dùng để lập trình (IDE), gọi là *trình biên dịch* (*compiler*) sẽ đảm đương việc này. Nó được gọi là *trình biên dịch C* vì nó biên dịch chương trình C (nói gọn là *mã C*) ra chương trình ở dạng ngôn ngữ máy (nói gọn là *mã máy*). Đúng ra, có nhiều trình biên dịch C cho nhiều loại/hệ máy khác nhau vì mỗi loại/hệ máy có ngôn ngữ riêng. Ta sẽ không đi xa hơn, nhưng bạn cần phải nhớ: *mã C cần được biên dịch ra mã máy trước khi chạy*. Như vậy thuật ngữ "*gõ và chạy chương trình*" phải được hiểu đúng là: bạn (lập trình viên) gõ (viết) mã C, sau đó trình biên dịch C biên dịch mã C thành mã máy, sau đó máy chạy (thực thi) mã máy[4]. Ba công đoạn này là tách bạch (và dĩ nhiên phải được thực hiện đúng trình tự trên), tuy nhiên các IDE thường có chức năng "biên dịch và chạy" thực hiện cả hai công đoạn sau[5]. Thật ra để máy có thể thực thi mã máy thì cần những hệ thống khác nữa mà quan trọng nhất là *hệ điều hành*. Ta sẽ gọi chung hệ thống thực thi này (máy + hệ điều hành + …) là *môi trường thực thi* và mã máy được gọi là *mã thực thi* hay *chương trình thực thi*. Hình 1.6.1 minh họa qui trình "gõ, biên dịch và chạy" này.

Tôi đã yêu cầu bạn phải tách bạch việc biên dịch và chạy vì mã C ta viết ra có thể bị *lỗi lúc biên dịch* hoặc *lỗi lúc chạy*. Nếu bạn thấy rõ điều này thì bạn đương nhiên hiểu rằng một lỗi biên dịch thì không thể là lỗi lúc chạy vì khi mã C bị lỗi biên dịch thì nó không được biên dịch ra mã máy nên không chạy được. Cũng vậy, một chương trình C không có lỗi biên dịch nào cũng có thể có lỗi lúc chạy.

Những lỗi nào là lỗi biên dịch? Để biên dịch thì trước hết compiler sẽ *phân tích* mã C để hiểu nội dung, tức là công việc của chương trình. Làm sao để phân tích một văn bản viết theo một ngôn ngữ nào đó để hiểu nội dung của văn bản đó. Văn bản là dãy các kí hiệu được ghép nối theo các qui tắc của một ngôn ngữ để mô tả một nội dung nào đó mà ta còn gọi là *ngữ nghĩa* của văn bản.

- Tập các kí hiệu được phép của một ngôn ngữ còn gọi là *bộ kí tự* của ngôn ngữ đó. Văn bản không được chứa các kí tự không có trong bộ

[3] Thật ra, thuở sơ khai, người ta phải viết chương trình bằng ngôn ngữ máy, và ngày nay, cũng có một số cao thủ lập trình bằng hợp ngữ cho các chương trình đặc biệt.
[4] Thật ra có vài thứ bí ẩn tham gia nữa mà ta sẽ rõ sau.
[5] Chẳng hạn chức năng *Compile & Run* (phím tắt Ctrl+F5 trong Microsoft Visual Studio hay F11 trong Dev-C++).

kí tự. Nếu vi phạm thì gọi là *lỗi kí tự*. Vậy bộ kí tự của C là gì? Nói chung là các kí tự thông thường gõ được từ bàn phím[6]. Lưu ý là bộ kí tự C không có các kí tự có dấu tiếng Việt[7]. Chẳng hạn chương trình có chứa kí tự @ thì có lỗi kí tự do @ không có trong bộ kí tự C[8].

- Các kí tự được gom lại thành các đơn vị gọi là *từ*. Giữa các từ thường có kí hiệu phân cách (như kí tự khoảng trắng) hoặc không cần nếu dễ dàng phân biệt[9]. Các từ được xác định theo qui tắc tạo từ (còn gọi là *từ pháp*) và được phân loại theo ngữ nghĩa (nội dung mà từ mô tả) gọi là *từ loại*. Khi vi phạm qui tắc tạo từ, văn bản bị *lỗi từ vựng*. Chẳng hạn câu "tổi ăn cơm" có lỗi từ vựng do các kí tự t, ổ và i không thể ghép thành từ (không có từ tổi trong tiếng Việt).

  Các từ trong C gọi là *token* và gồm 6 từ loại chính. Bạn hãy xem Phụ lục A.5 để biết và nắm vững các từ loại của C nhé. Dòng 4 của chương trình trên có lỗi từ vựng là `1oo`. Chữ số `0` bị gõ nhầm thành chữ cái o. Đây không phải là lỗi kí tự vì cả `0` lẫn o đều thuộc bộ kí tự của C. Đây là lỗi từ vựng, cụ thể, ta đã vi phạm qui tắc tạo *hằng số* đúng (phải ghép các chữ số lại, không được ghép kí tự không phải chữ số sau chữ số). Lưu ý, nếu bạn viết tách ra `1 oo` thì không có lỗi từ vựng: ta có 2 token, `1` là *hằng số nguyên* và `oo` là *định danh*. Hãy sửa lại để được hằng số đúng là `100`. Dòng 5 của chương trình trên có lỗi từ vựng là thiếu dấu `"` đóng trong việc tạo *hằng chuỗi*. Thêm vào để được hằng chuỗi đúng là `"What's your name?"`.

- Các từ, sau đó, được gom lại theo qui tắc thành các cấu trúc cao hơn như cụm từ, câu, đoạn văn, văn bản. Những qui tắc đó được gọi chung là *cú pháp*. Cú pháp, như vậy, là cách lắp ghép các từ theo qui tắc hay khuôn mẫu. Bạn đã gặp khuôn mẫu của chương trình, khuôn mẫu khai báo biến, khuôn mẫu định nghĩa hàm, khuôn mẫu của lệnh nhập, xuất, gán, khuôn mẫu của biểu thức.

  C không giống ngôn ngữ tự nhiên như tiếng Việt lắm nhưng cũng có thể hình dung: biểu thức là cụm từ, câu lệnh hay khai báo biến là câu, nhiều câu lệnh liên quan viết kề nhau là đoạn văn và toàn bộ chương trình là văn bản. Thuật ngữ văn bản cũng được dùng với nhiều mức độ từ đơn giản đến phức tạp. Nó có thể ngắn gọn như một bài văn nhỏ hay một lá đơn. Nó cũng có thể phức tạp

[6] Xem Phụ lục A.5 để biết bộ kí tự của C.

[7] Tùy phiên bản C và compiler mà bộ kí tự có khác nhau, C99 có bộ kí tự Unicode, cho phép cả các kí tự có dấu tiếng Việt.

[8] Hằng chuỗi, `"..."`, được phép chứa các kí tự không có trong bộ kí tự C, chẳng hạn hằng chuỗi `"abc@gmail.com"` là hợp lệ.

[9] Tiếng Việt có đơn vị nữa gọi là *tiếng*, như từ "gia đình" có 2 tiếng, điều này làm cho việc xác định các từ rất khó khăn trong tiếng Việt.

như cả một cuốn sách gồm nhiều chương mà mỗi chương lại có thể xem là văn bản gồm nhiều phần mà mỗi phần cũng có thể xem là văn bản. Chương trình C nếu phức tạp thì sẽ gồm nhiều *module* (mà ta có thể xem như chương) mà mỗi module có thể gồm nhiều *hàm* (mà ta có thể xem như phần).

Khi văn bản vi phạm các qui tắc cú pháp thì ta gọi là *lỗi cú pháp*. Chẳng hạn, "ăn cơm tôi" không là câu đúng của tiếng Việt, nó vi phạm qui tắc tạo câu (qui tắc Chủ ngữ-Vị ngữ). Dòng 6 của chương trình trên có lỗi cú pháp là thiếu dấu `;`. Nhớ rằng cú pháp của *lệnh đơn*[10] là có dấu `;` ở cuối. Thêm dấu `;` vào để được lệnh đúng cú pháp là: `scanf("%s", &name);`

- Bộ kí tự, từ pháp, cú pháp qui định *hình thức* của văn bản. Khi văn bản không vi phạm các qui tắc của kí tự, từ pháp, cú pháp thì nó là văn bản đúng về hình thức và nó sẽ mô tả một *nội dung* mà ta gọi là *ngữ nghĩa* của văn bản. Tên biến, tên hàm ám chỉ đến biến, hàm tương ứng; toán tử +, - mô tả các phép toán cộng, trừ; … Ngữ nghĩa của biểu thức là giá trị của biểu thức đó. Ngữ nghĩa của câu lệnh là sự thực thi tương ứng, như lệnh gán sẽ tính giá trị của biểu thức bên vế phải (là ngữ nghĩa của biểu thức) rồi bỏ giá trị vào biến bên vế trái (là ngữ nghĩa của tên biến). Ngữ nghĩa của hàm là công việc của hàm đó còn ngữ nghĩa của chương trình là toàn bộ công việc của chương trình.

  Cũng lưu ý là ngữ nghĩa còn phụ thuộc vào *ngữ cảnh* xuất hiện, như: ngữ nghĩa của toán tử / là phép chia số nguyên nếu cả hai toán tử là số nguyên và là phép chia số thực nếu ít nhất một trong hai toán tử là số thực[11] hay ngữ nghĩa của tên biến là vùng nhớ của biến (cái hộp) nếu tên biến dùng trong lệnh gán (bên trái dấu =) và là giá trị của biến (dữ liệu bên trong cái hộp) nếu tên biến dùng trong biểu thức[12].

  Khi mà nội dung của văn bản có vấn đề gì đó (mâu thuẫn, không phù hợp, không đúng đắn, …) thì văn bản được nói là có *lỗi ngữ nghĩa*. Chẳng hạn "cơm ăn tôi" là một câu tiếng Việt tốt về hình thức nhưng có vấn đề về nội dung, nó có lỗi ngữ nghĩa[13]. Ở Dòng 2 của chương trình trên, tên hàm `main` bị gõ nhầm thành `Main`. Không

[10] Ta sẽ định nghĩa chính xác lệnh đơn là gì. Tạm thời ta hiểu đó là lệnh "đơn giản" như lệnh gán, lệnh nhập, xuất.
[11] Phép chia số nguyên và phép chia số thực là hai thao tác hoàn toàn khác nhau trên máy.
[12] Dấu = là *toán tử* chứ không phải là *dấu câu* vì nó mô tả một phép toán (thao tác), là phép gán.
[13] Mặc dù trong một ngữ cảnh nào đó, phim hoạt hình chẳng hạn, thì câu đó vẫn tốt về ngữ nghĩa.

có vấn đề gì với một hàm tên `Main` cả: `main` và `Main` đều là các định danh hợp lệ và có thể dùng làm tên của hàm. Nhưng như vậy thì chương trình trên không có hàm `main`[14] và do đó chương trình không thể chạy được (cho dù compiler chịu biên dịch ra mã máy).

Cũng lưu ý, một lệnh như `int a = 1.1;` là không hợp lý về ngữ nghĩa vì `1.1` là một giá trị thực trong khi a là biến nguyên, nó chỉ có thể chứa giá trị nguyên. Thực sự, trong nhiều ngôn ngữ như Java, C# thì đây là lỗi ngữ nghĩa, nhưng trong C đây không là lỗi vì C tự động chuyển 1.1 thành giá trị nguyên 1 (bằng cách cắt bỏ phần lẻ).

Tất cả các lỗi hình thức (kí tự, từ vựng, cú pháp) đều là *lỗi* (*error*) biên dịch. Các bất thường (hay không hợp lý) về ngữ nghĩa nếu nặng thì là lỗi biên dịch và nhẹ thì là các *cảnh báo* (*warning*). Chẳng hạn nếu một biến được dùng mà chưa khai báo thì là lỗi, còn như một biến được khai báo mà không dùng thì là một bất thường về ngữ nghĩa nhưng thường chỉ được compiler cảnh báo chứ không xem là lỗi[15]. Bất thường vì nếu không cần thì ta đã không dùng, mà không dùng thì khai báo làm gì? Tuy nhiên nó không gây ảnh hưởng gì (trừ chuyện tốn một ít bộ nhớ). Cũng lưu ý là một số bất thường về ngữ nghĩa rất khó phân tích để thấy nên không được compiler cảnh báo.

Trong ngôn ngữ tự nhiên ta hay nghe nói đến *lỗi chính tả*. Trong C không có khái niệm lỗi chính tả. Đây là thuật ngữ rất mơ hồ, ý nói lỗi gây nên do viết sai (mà thường là viết nhầm). Mà lỗi nào không do viết sai (hay viết không đúng) chứ?! Như vậy lỗi nào cũng là lỗi chính tả. Bạn có thể nói rằng do tên hàm `main` bị gõ sai (hay gõ nhầm) thành `Main` nên đây là lỗi chính tả. Mơ hồ! Đây là lỗi ngữ nghĩa như đã thấy ở trên nhé. Tương tự, những người mới bắt đầu lập trình cũng thường gõ nhầm tên hàm `printf` thành `print`, mà nguyên nhân có lẽ do print là từ tiếng Anh nghĩa là in hay xuất, còn printf thậm chí không phải là một từ đúng chính tả tiếng Anh[16]. Những người đó như vậy hay gọi đó là lỗi chính tả, tuy nhiên đó là lỗi ngữ nghĩa: bạn gọi một hàm (`print`) mà hàm đó chưa được định nghĩa[17].

Nếu có lỗi biên dịch thì compiler không chịu biên dịch ra mã máy nên không có chương trình để chạy. Hơn nữa những lỗi này (cùng với các cảnh báo) là dễ sửa vì compiler sẽ thông báo chỗ lỗi và hỗ trợ ta sửa lỗi (xem Phụ lục A.2 để biết cách sửa lỗi biên dịch). Nếu chương trình không có lỗi biên

[14] Nhớ rằng C phân biệt chữ hoa chữ thường và mọi chương trình C đều cần có đúng một hàm `main`.

[15] Điều này còn tùy thuộc compiler và tùy thuộc cấu hình mức độ khắc nghiệt:) của compiler, ở mức ít khắc nghiệt nhất thì có thể không có cả cảnh báo.

[16] `printf` là viết tắt của print with format (in có định dạng).

[17] Còn nếu như hàm `print` đã được định nghĩa với khai báo đầu vào phù hợp thì chương trình lại không có lỗi ngữ nghĩa mà là *lỗi logic*: ý là gọi `printf` nhưng lại gọi nhầm `print`.

dịch (dù có warning hay không) thì compiler sẽ biên dịch ra mã thực thi và môi trường thực thi có thể chạy mã này. Nhưng như vậy không có nghĩa là chương trình không còn lỗi. Chương trình (đã ở dạng mã thực thi) khi chạy vẫn có thể có sự cố, bất thường hay chạy không đúng mong đợi. Những lỗi này gọi là *bug*[18]. Những bug mà khi gặp làm cho chương trình bị sụp, không chạy được nữa thì gọi là *lỗi thực thi* (runtime error). Chẳng hạn lỗi chia cho 0 mà ta đã gặp là dạng lỗi thực thi hay gặp khi chương trình có thực hiện phép chia. Hay như trong chương trình trên, giả sử ở Dòng 6 ta quên gõ biến `name` thành:

```
scanf("%s");
```

thì chương trình cũng bị lỗi runtime, nó cũng bị sụp như khi chia cho 0 vậy. Những lỗi runtime này, mặc dù, gây xấu hổ[19] nhưng dù sao cũng nhận biết được và chương trình sẽ dừng (nghĩa là không gây hậu quả thêm nữa) khi gặp lỗi này.

Những bug mà không gây sụp chương trình (nghĩa là không phải lỗi runtime) thì gọi là *lỗi logic*. Với lỗi logic, chương trình chạy có vẻ bình thường nhưng lại không chạy đúng như mong đợi (đúng hơn là chạy bất thường nhưng không có dấu hiệu rõ ràng). Chẳng hạn trong chương trình trên, giả sử ở Dòng 7 thay vì gõ `%s` ta gõ nhầm thành `%d` như sau:

```
printf("Hello %d",  name);
```

thì chương trình vẫn chạy bình thường nhưng lại xuất ra lời chào đến một cái tên toàn là số! Nếu như lỗi logic này cũng dễ nhận thấy (thường thì tên người không phải là dãy số) thì chương trình đúng đến giờ (đã sửa tất cả các lỗi ở trên):

Mã 1.6.2 – Chương trình Hello đã sửa nhiều lỗi:)

```
1  #include <stdio.h>
2  int main()
3  {
4      char name[100];
5      printf("What's your name?");
6      scanf("%s", &name);
7      printf("Hello %s", name);
8  
9      return 0;
10 }
```

---

[18] Nghĩa đen là sâu bọ. Ở đây thì là lỗi trong chương trình đã chạy được. Nghĩa này có lẽ từ câu tục ngữ "một con sâu làm rầu nồi canh" ?! (là tôi cố ý đoán vậy thôi chứ chắc hổng phải:))

[19] Nó là dấu hiệu cho thấy công lực của lập trình viên còn yếu.

vẫn còn lỗi logic. Bạn có biết đó là lỗi gì không? Chạy thử vài lần và nếu may mắn bạn sẽ thấy rằng nếu người dùng nhập tên có khoảng trắng (chẳng hạn `Dennis Ritchie`) thì chương trình chỉ xuất ra lời chào đến phần tên trước khoảng trắng (chẳng hạn `Dennis`). Rõ ràng đây không phải là điều mà người viết chương trình mong đợi ở chương trình nên đây là lỗi logic. Ta có thể sửa lỗi này bằng cách thay vì dùng `scanf("%s", &name)` ta dùng `gets(name)`. Kĩ năng tìm và *sửa bug* (*debug*), như vậy, là kĩ năng rất quan trọng mà ta sẽ phải trui rèn thường xuyên.

Hình 1.6.1 – Hình minh họa qui trình "gõ, biên dịch và chạy chương trình"

Bài toán
Công việc
Phân tích
Thiết kế
Thuật toán
Các bước làm
Lập trình viên
Viết mã
- lỗi từ vựng
- lỗi cú pháp
- lỗi ngữ nghĩa
Chương trình
C
Trình biên dịch
Biên dịch
Dữ liệu
(input)
Môi trường thực thi
Có lỗi biên dịch
Không lỗi
Nhập liệu
Tương tác
Thực thi
Chương trình
thực thi
Có bug
Không bug
Thư viện
Người dùng
Bất thường
Kết xuất
(output)
- lỗi runtime
- lỗi logic
- lỗi tìm ẩn

Sau khi sửa như trên (thay Dòng 6 thành `gets(name);`) thì chương trình đã hết lỗi chưa? Không còn *lỗi tường minh* (lỗi rõ ràng) nhưng vẫn còn *điểm*

yếu hay *lỗ hổng* mà ta có thể xem là *lỗi tìm ẩn*. Cụ thể, nếu người dùng gõ tên quá dài (quá 100 kí tự) thì chuỗi nhập vào sẽ quá sức chứa của biến `name` (đã được khai báo là chứa không quá 100 kí tự), hậu quả có thể khác nhau (tùy vào nhiều thứ khác nữa): chương trình có thể bị sụp (như vậy đó là lỗi runtime) hoặc chương trình có dấu hiệu bất thường nào đó (có thể xem là lỗi logic) hay thậm chí không có biểu hiện gì nhưng một vài bất thường bên trong đã xảy ra. Ta có thể đổ lỗi rằng tại người dùng không theo yêu cầu là nhập tên không quá 100 kí tự, thậm chí cẩn thận hơn chương trình có thể in ra thông báo nhập là nhập không quá 100 kí tự, nhưng không phải người dùng nào cũng là người tốt:), có những người dùng chỉ muốn phá hoại chương trình hoặc khai thác những điểm yếu của chương trình để thu lợi bất chính. Đây là một chủ đề rất lớn và khó mà ta sẽ tìm hiểu một phần trong các bài sau. Ở đây ta cần biết rằng các hàm như `printf`, `scanf`, `gets` (dù là các hàm trong thư viện chuẩn) là các hàm có nhiều điểm yếu[20].

Không có lỗi (hay "*sạch lỗi*") là tiêu chí đầu tiên khi viết chương trình. Như bạn đã thấy, điều này không hề dễ tí nào. Ta cần loại bỏ mọi lỗi biên dịch và hạn chế tối đa các lỗi khác. Một chương trình không có lỗi thì đúng về cả hình thức lẫn nội dung. Ngoài ra, cũng như khi viết một bài văn, hai tiêu chí nữa mà ta cũng cần quan tâm khi viết mã C là *rõ ràng* và *ngắn gọn*. Rõ ràng nghĩa là trình bày đẹp, dễ đọc, dễ hiểu; ngắn gọn nghĩa là viết cô đọng, súc tích. Tại sao cần viết đẹp và rõ ràng? Ai là người đọc mã C của ta? Dưới mắt compiler thì mã C chẳng qua là dãy các token[21]. Mọi mã C đều có thể viết chỉ trên một dòng mà không làm rối mắt compiler. Tuy nhiên, còn những người khác nữa cũng đọc mã C. Trước hết là chính bản thân lập trình viên, người đã viết mã, là người thường xuyên đọc đi đọc lại mã, lúc viết hay vài tháng, năm sau đó khi cần chỉnh sửa hay phát triển chương trình. Thứ hai, nếu chương trình lớn thì nó có thể được viết bởi cả một nhóm nhiều lập trình viên, người khác phải hiểu mã bạn viết và ngược lại, bạn phải hiểu mã người khác viết. Thứ ba, mã bạn viết có thể cho cả cộng đồng đủ dạng lập trình viên. Chẳng hạn như tôi, cần viết code đẹp, rõ ràng để bạn, người mới bắt đầu học có thể dễ dàng hiểu được.

Cũng lưu ý là tiêu chí rõ ràng và ngắn gọn thường mâu thuẫn nhau: quá ngắn gọn có thể khó hiểu và ngược lại để dễ hiểu thì đôi khi phải viết dài. Bạn có hiểu nổi chương trình C sau đây không?[22]

Mã 1.6.3 – Một chương trình C bí hiểm

---

[20] Microsoft C++ compiler cung cấp các hàm an toàn hơn là `printf_s`, `scanf_s`, `gets_s`. Các hàm này không nằm trong thư viện chuẩn cũ.

[21] Ở mức thấp hơn, dãy các kí tự sẽ được gom lại thành dãy các token.

[22] http://www.ioccc.org/1990/baruch.c. Đây là chương trình của Doron Osovlanski và Baruch Nissenbaum, đạt giải "Best Small Program" của cuộc thi "The International Obfuscated C Code Contest" năm 1990.

```
v,i,j,k,l,s,a[99];
main()
{
    for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j
    <s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j
    )&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||
    (i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

Chương trình trên giải bài toán "8-hậu": đặt 8 quân hậu vào bàn cờ vua 8×8 sao cho không có quân hậu nào tấn công quân hậu nào[23]. Chạy chương trình trên, nhập 8, thì chương trình sẽ in ra tất cả các khả năng đặt, chẳng hạn một trong số đó là:

```
 12345678
1 # # # Q
2# #Q# #
3Q# # # #
4# Q # #
5 # # Q #
6#Q# # #
7 # # #Q#
8# # Q #
```

Mã C trên cô đọng đến kinh ngạc và cũng khó hiểu biết bao. Nó giống như một văn bản bí hiểm của người ngoài hành tinh!

Với người mới bắt đầu học viết chương trình thì nên đặt tiêu chí *code đẹp* lên hàng đầu. Dĩ nhiên code đẹp không phải là viết chữ đẹp mà là viết ngăn nắp, dễ nhìn. Để như vậy, ta cần dùng các *kí tự trắng* (space, tab, xuống dòng) phù hợp. Đây không phải là token mà là các kí tự giúp phân cách token và định dạng mã. Bạn có thể dùng chúng, bao nhiêu cũng được, giữa các token mà không ảnh hưởng đến việc biên dịch ra mã thực thi. Nói chung là "cách ra, thụt vào, xuống dòng" cho phù hợp.

Vì mã C thường rất cô đọng và không dễ để hiểu, thậm chí, với cả người đã viết mã đó[24]. Cho nên, hãy cố gắng *ghi chú* thêm vào mã, nhất là những chỗ khó hiểu. Những ghi chú thêm này không tham gia vào mã (nghĩa là có hay không có nó đều không ảnh hưởng đến việc biên dịch ra mã thực thi) mà là để người đọc (có thể là chính người viết mã) dễ hiểu nội dung của mã hơn. Trong C bạn có thể ghi chú bằng hai cách:

- Những ghi chú dài, thường là nhiều dòng, có thể để trong cặp dấu /* và */. Chẳng hạn đầu chương trình thường có ghi chú về tác giả,

[23] Thật ra chương trình trên giải bài toán tổng quát "*N*-hậu".

[24] Có thể, sau một tháng, bạn đọc lại mã của mình viết và lầm bầm rằng "thằng nào code mà đọc không hiểu gì hết":)

ngày viết, mục đích, ý nghĩa của chương trình. Đầu mỗi hàm cũng thường có ghi chú về công việc của hàm đó, ý nghĩa của các đầu vào và giá trị trả về của hàm.

- Những ghi chú ngắn trên một dòng có thể đặt sau dấu `//`[25]. Những ghi chú về ý nghĩa của biến hay công việc của một lệnh nào đó thường ở dạng này.

Ghi chú cũng thường được dùng để chứa các đoạn mã viết nháp hoặc chứa các đoạn mã cũ mà bạn không muốn xóa. Cũng lưu ý là tránh viết những ghi chú nhảm, không cần thiết:) Chẳng hạn tránh ghi chú đầu chương trình:

```
/* Day la chuong trinh cua tui
 * Viet ngay hom nay
 * Viet tren may nay
 */
```

hay ghi chú đầu hàm `main`:

```
// Day la ham main
```

hay ghi chú cho lệnh gán:

```
a = 10 + b; // a se chua tong cua 10 voi gia tri cua b
```

Việc ghi lại và tổ chức các thông tin hữu ích đi kèm chương trình được gọi là *sưu liệu*. Đây là việc rất quan trọng mà ta sẽ tìm hiểu thêm sau.

Để mã dễ hiểu bạn cũng nên đặt tên biến, tên hàm gợi nhớ, tức là tên gợi ý cho mục đích, ý nghĩa của nó. Bạn cũng nên đặt tên biến bằng một danh từ vì biến mô tả dữ liệu (tức là cái gì đó), đặt tên hàm bằng động từ vì hàm mô tả công việc (tức là làm gì đó). Cũng nên dùng các qui ước đặt tên thông thường trong Toán, chẳng hạn diện tích thì là *S*, thể tích thì là *V*, hằng số π thì là *pi*, … Các tên cũng nên ngắn gọn và nên dùng từ tiếng Anh hơn là tiếng Việt.

Bạn cũng tránh lạm dụng *kĩ thuật cá nhân*: những mẹo (hay kĩ thuật) viết mã quá cô đọng hay những toan tính để mã viết ra biên dịch thành mã máy tối ưu (nghĩa là chạy nhanh). Những mã như vậy thường rất khó hiểu. Một bài thơ, hàm súc đến mức không hiểu được, thì không hữu ích cho nhiều người. *Hãy code đẹp, rõ ràng, tự nhiên và tránh rườm rà!*

Bạn thấy hơi choáng? thấy hơi lung tung, mơ hồ? Đó chính là mục đích của bài này, tôi đã cố ý hù dọa bạn một chút sau những bài đầu quá dễ, để bạn thấy rằng C là gì đó vừa dễ vừa khó, vừa đơn giản vừa phức tạp, vừa rõ ràng vừa mơ hồ, rất tự nhiên nhưng cũng đầy bất trắc:). Tuy nhiên, chắc

[25] Cách ghi chú này có nguồn gốc từ C++.

chắn là bạn sẽ tự thấy mọi thứ đơn giản, rõ ràng và tự nhiên khi công lực tăng cao[26].

BÀI TẬP

Bt 1.6.1 Phân tích từ vựng đoạn mã C sau đây:

```
int main()
{
    printf("10^2 = %d", 10 * 10);

    return 0;
}
```

(*Xem Hướng dẫn giải bên dưới.*)

Bt 1.6.2 Tương tự Bài tập 1.6.1, phân tích từ vựng các đoạn mã C sau:

a) Mã 1.3.1 của Bài 1.3
b) Mã 1.5.4 của Bài 1.5
c) Mã 1.5.5 của Bài 1.5

Bt 1.6.3 Xác định các biểu thức, khai báo, câu lệnh (với loại lệnh tương ứng) trong các đoạn mã C sau:

a) Mã 1.2.2 của Bài 1.2
b) Mã 1.4.1 của Bài 1.4
c) Mã 1.5.1 của Bài 1.5
d) Mã 1.5.5 của Bài 1.5

Bt 1.6.4 Cho biết đoạn mã C sau có lỗi hay không? Nếu có hãy chỉ ra lỗi (lỗi chỗ nào, lỗi loại gì) và đưa ra cách sửa lỗi.

```
int d = 10
printf("Gia tri cua d la: %d", &d);
```

(*Xem Hướng dẫn giải bên dưới.*)

Bt 1.6.5 Tương tự Bài tập 1.6.4 cho các đoạn mã C sau. Giả sử `n`, `m` là các biến nguyên (`int`) đã được khai báo trước đó.

```
a) Scanf("d", n)
b) scanf("%d%f", &n m);
c) printf("%dd %lf", n, m);
d) printf("Tong cua %d va %d la %d"\n, n, m, n-m);
e) n == 10
   printf("Gia tri cua n la: ", n);
f) If(n =< 10);
       printf("n nho hon hoac bang 10");
g) if(n = m)
```

---

[26] Đừng để bị tôi hù dọa mà bỏ cuộc nhé. Hãy tiếp tục luyện lên tầng cao hơn.

```
    printf("n bang m");
h) / * Xuat ra cac so tu 1 toi n **/
    for(int i = 1, i <= n, i = i - 1)
        printf("%d", i);
```

Bt 1.6.6 Sau đây là một chương trình C hợp lệ:

```
#include <stdio.h>
int main(){int n;scanf("%d", &n);do
{printf("%d",n%10);n
=n/10;}while(n>0);return
0;}
```

Bạn có hiểu chương trình này không? Bạn có biết nó thực hiện công việc gì không? Hãy định dạng lại mã cho đẹp, cho dễ nhìn. Chạy thử để hiểu công việc của nó. Thêm các thông báo nhập xuất cho dễ hiểu. Thêm các ghi chú vào những chỗ cần thiết để người khác đọc mã có thể dễ hiểu hơn.

Hướng dẫn giải Bài tập 1.6.1

Phân tích từ vựng là bước đầu tiên phải làm để phân tích văn bản của bất kì ngôn ngữ nào (tiếng Việt, tiếng Anh, tiếng C, tiếng C++, ...). Đó là việc xác định các *từ* (C gọi là *token*) và chỉ ra *từ loại* của chúng. Mỗi token là dãy các kí tự liền nhau (không bị phân cách bởi khoảng trắng[27]) nhưng một dãy các kí tự liền nhau có thể gồm nhiều token. Bạn cần nhận diện được các token từ qui tắc của 6 loại token trong C[28] (xem Phụ lục A.5). Các IDE C (hay các Editor có hỗ trợ C) cũng thường dùng màu sắc khác nhau cho các loại token khác nhau. Do đó, chỉ cần gõ mã C trong IDE, hầu như bạn sẽ thấy rõ[29]. Đoạn mã trên gồm 18 token theo thứ tự với từ loại tương ứng như sau:

Bảng 1.6.1 – Bảng phân tích từ vựng đoạn mã C của Bài tập 1.6.1

STT	Token	Loại oken	STT	Token	Loại token
1	int	Từ khóa	10	10	Hằng số
2	main	Định danh	11	*	Toán tử
3	(	Dấu câu	12	10	Hằng số
4	)	Dấu câu	13	)	Dấu câu
5	{	Dấu câu	14	;	Dấu câu
6	printf	Định danh	15	return	Từ khóa

[27] Trừ khoảng trắng trong hằng chuỗi.
[28] Mà làm việc này rất dễ so với các ngôn ngữ như tiếng Việt, tiếng Anh.
[29] Đoạn mã của bài tập này được chép lại với định dạng màu từ Editor có hỗ trợ C là *Notepad++*. Như bạn thấy: các định danh có màu đen, các từ khóa có màu tím hoặc xanh (tím cho từ khóa liên quan đến dữ liệu và xanh cho từ khóa liên quan đến lệnh), các hằng số có màu cam, ... Cũng nhớ là bạn có thể cấu hình các Editor (hay IDE) để chọn màu mong muốn cho các loại token.

7	(	Dấu câu	16	0	Hằng số
8	"10^2 = %d"	Hằng chuỗi	17	;	Dấu câu
9	,	Dấu câu	18	}	Dấu câu

Hướng dẫn giải Bài tập 1.6.4

Lỗi trong mã C có thể chia ra 2 nhóm chung là *lỗi biên dịch* và *bug*. Khi có lỗi biên dịch, mã không thể thực thi được (chạy được). Khi biên dịch, tất cả các IDE đều chỉ ra lỗi biên dịch trong mã (chỗ nào và mô tả của lỗi) nếu có nên lập trình viên dễ dàng xác định và sửa lỗi biên dịch. Tuy nhiên dù mã chạy được (tức là không có lỗi biên dịch) thì nó vẫn có thể có bug. Đó là các bất thường (về ngữ nghĩa, ý nghĩa, mục đích , ...) khi thực thi mã. Việc phát hiện (và sửa) các bug, như vậy, khó khăn hơn nhiều.

Hiển nhiên là đoạn mã trên không thể chạy độc lập được mà cần gắn vào chương trình như sau:

```
#include <stdio.h>
int main()
{
    int d = 10
    printf("Gia tri cua d la: %d", &d);

    return 0;
}
```

Biên dịch chương trình trên (bằng IDE nào đó) bạn sẽ thấy thông báo lỗi biên dịch. Cụ thể đoạn mã trên có 1 lỗi biên dịch là thiếu dấu ; sau hằng số 10 để tạo thành khai báo với khởi tạo. Ta sửa lại như sau:

```
int d = 10;
printf("Gia tri cua d la: %d", &d);
```

Đến đây thì đoạn mã chạy được (không còn lỗi biên dịch) nhưng khi chạy thì có bất thường. Cụ thể, chạy thử bạn sẽ thấy một con số lạ quắc (không phải 10) được xuất ra dù đoạn mã trên dự định xuất ra giá trị trong biến d mà d đang chứa 10 nên kết quả xuất ra phải là 10. Như vậy đoạn mã trên có bug. Quan sát kĩ ta thấy dư dấu & trước biến d trong lệnh xuất[30]. Vậy đoạn mã được sửa lại như sau sẽ không còn lỗi:

```
int d = 10;
printf("Gia tri cua d la: %d", d);
```

[30] Dấu & trước tên biến chỉ dùng trong lệnh nhập.

BÀI 1.7

Trong khi máy dùng *số*[1] để biểu diễn/lưu trữ dữ liệu và liên lạc thì chúng ta dùng *chuỗi* để mô tả/lưu trữ thông tin/tri thức và giao tiếp. Chuỗi trong bất kì ngôn ngữ nào cũng được tạo nên từ các *kí hiệu* (symbol) hay *kí tự* (character)[2] gọi là *tập kí hiệu* của ngôn ngữ. Lưu ý rằng thuật ngữ tập kí hiệu được dùng với nghĩa rộng để chỉ tất cả các kí hiệu dùng trong ngôn ngữ. Như vậy nó bao gồm các *chữ cái* (letter) trong *bảng chữ cái* (alphabet)[3], các *kí số* (digit), các dấu như *dấu câu* (punctuation symbol) và các kí hiệu khác.

Khi chúng ta dùng máy làm phương tiện tổ chức/lưu trữ/xử lý dữ liệu/thông tin/tri thức và truyền thông thì ta cần cách nào đó để đưa tập kí hiệu vào máy gọi là *cách mã kí hiệu*. Một cách tự nhiên, máy sẽ dùng các số (nguyên không âm) để mã các kí hiệu[4]: mỗi kí hiệu được xác định bằng một con số được gọi là *mã số* (code hay code point) của kí hiệu đó. Tập các mã (số) của các kí hiệu được gọi là *bảng mã*. Có nhiều tiêu chí cho việc lựa chọn bảng mã (tức là lựa chọn mã số cho các kí hiệu) như: thuận lợi cho xử lý, mã được nhiều kí hiệu, tiết kiệm không gian lưu trữ và truyền thông, … nhưng quan trọng nhất là *tính phổ biến* của bảng mã. Khi nhiều tổ chức/cá nhân/máy … cùng dùng chung một bảng mã thì việc trao đổi thông tin sẽ thống nhất và thông suốt.

Bảng mã được dùng phổ biến nhất trên các hệ máy/hệ điều hành/hệ thống phần mềm khác nhau là *bảng mã ASCII*. Bảng mã này dùng 128 số nguyên không âm đầu tiên (0, 1, 2, …, 127) làm mã số của 128 kí hiệu được lựa chọn khác nhau mà chủ yếu là các "kí tự của Mỹ"[5] gồm:

- 32 mã số đầu tiên (0, 1, …, 31) dùng cho 32 *kí hiệu điều khiển* trong đó quan trọng nhất là kí tự NULL (mã 0) dùng làm kí hiệu đánh dấu kết thúc chuỗi.
- 95 mã số tiếp theo (32, 33, …, 126) dùng cho 95 *kí hiệu in được*[6] (hay hiển thị được, viết được) gồm:

---

[1] Số nguyên, số thực với các kích thước khác nhau.

[2] Trong một số trường hợp, thuật ngữ kí hiệu được dùng với nghĩa rộng hơn thuật ngữ kí tự.

[3] Chẳng hạn bộ chữ cái tiếng Việt có 29 chữ cái (chưa kể chữ cái có dấu) là A/a, Ă/ă, Â/â, …, X/x, Y/y, bộ chữ cái tiếng Anh có 26 chữ cái là A/a, B/b, C/c, …, X/x, Y/y, Z/z.

[4] Đây là qui tắc quan trọng, xuyên suốt: máy dùng số (đặc biệt là số nguyên) để "mã hóa" các dạng dữ liệu/thông tin khác nhau.

[5] Tên gọi đúng hơn là *US-ASCII* cho thấy nguồn gốc ra đời của nó.

[6] Thuật ngữ kí tự khi dùng với nghĩa hẹp là để chỉ các kí hiệu in được.

    - Kí tự SPACE (mã 32) là kí tự "khoảng trắng" dùng mô tả khoảng trắng giữa các từ[7].
    - Các chữ cái tiếng Anh hoa A, B, …, Z với mã số 65, 66, …, 90.
    - Các chữ cái tiếng Anh thường a, b, …, z với mã số 97, 98, …, 122.
    - Các kí số thập phân 0, 1, …, 9 với mã số 48, 49, …, 57.
    - Các kí tự còn lại là các dấu câu như: !, ., ?, +, -, *, /, …
- Mã số cuối cùng (127) dùng cho kí tự điều khiển DEL.

Bạn hãy tham khảo bài viết ASCII trên trang Wikipedia[8] để biết rõ hơn. Ở trển cũng trình bày bảng mã ASCII rất đẹp:)

Bảng mã ASCII cũng được hầu hết các ngôn ngữ lập trình sử dụng, trong đó có C. Như ta đã thấy trong Bài 1.6, tập kí hiệu (hay bộ kí tự) của C là tập con của tập kí tự ASCII. Hơn nữa C cũng dùng mã (số) ASCII để biểu diễn và nhập/xuất kí tự/chuỗi. Vì 1 byte có thể biểu diễn được 256 số (từ số 0 đến số 255) nên C dùng kiểu số nguyên 1 byte để biểu diễn một kí tự[9]. Đó là kiểu `char`[10], mà như tên gọi cho thấy, nó dùng để chứa mã (số) ASCII của kí tự. Chẳng hạn đoạn mã sau:

```
char ch;
ch = 65;
printf("Ma ASCII cua ki tu %c la %d", ch, ch);
```

xuất ra kết quả là:

```
Ma ASCII cua ki tu A la 65
```

Trong đoạn mã trên thì biến `ch` kiểu `char` sẽ chứa số nguyên là mã ASCII của kí tự mà lệnh gán tiếp theo làm cho nó chứa số 65 là mã ASCII của kí tự A (hoa). Như vậy, "về mặt ý nghĩa" ta nói rằng `ch` chứa kí tự A nhưng "về mặt vật lý" thì `ch` chứa số nguyên 65. Lệnh xuất sau đó xuất ra chuỗi như minh họa ở trên với `%d` giúp xuất ra chuỗi số thập phân của giá trị 65 (`ch` chứa 65) và đặc biệt `%c` giúp xuất ra kí tự tương ứng với mã ASCII 65 (`ch` chứa 65 là mã ASCII của kí tự A). Cũng vậy đoạn mã sau sẽ giúp xuất ra tất cả các kí tự in được (mã ASCII từ 32 đến 126).

```
for(char ch = 32; ch <= 126; ch++)
    printf("%c", ch);
```

---

[7] Giữa các tiếng của Tiếng Việt cũng dùng khoảng trắng khi viết.
[8] https://en.wikipedia.org/wiki/ASCII
[9] Đúng ra thì chỉ cần dùng 7 bít là có thể biểu diễn được 128 mã số ASCII nhưng trên máy thì "chẵn byte" (8 bít) sẽ dễ xử lý hơn. (Đúng hơn là chẵn lũy thừa 2 của byte, tức là 1, 2, 4, 8 byte).
[10] Nếu hơi thắc mắc thì kiểu số nguyên `int` mà ta đã dùng trong các bài trước là kiểu số nguyên 4 byte.

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Như kết xuất trên cho thấy, sở dĩ các kí tự này được gọi là in được vì nó có hình ảnh hiển thị (nét chữ) đi kèm. Đặc biệt, kí tự khoảng trắng (SPACE) có hình ảnh hiện thị là khoảng trắng (nét chữ rỗng[11]). Lưu ý rằng ta phải phân biệt được kí tự với nét chữ của kí tự. Mỗi kí tự là duy nhất được xác định bởi mã của kí tự nhưng nét vẽ cho nó thì rất đa dạng do *font chữ* qui định. Chẳng hạn cùng chữ cái A hoa nhưng có người viết đẹp, người viết xấu, dễ nhìn/khó nhìn, chân phương/hoa mỹ, to/nhỏ, nghiêng/thẳng, đậm/nhạt, … Trên máy cũng vậy, nét chữ hiển thị cho kí tự sẽ khác nhau tùy theo font chữ được thiết lập[12].

Một truyền thống của C mà ta nên quen thuộc là việc dùng *hệ thống số cơ số* 16 để mô tả số nguyên (không âm), đặc biệt là mã kí tự[13]. Hệ thống số quen thuộc với ta, hệ cơ số 10 (*thập phân - decimal*), dùng 10 kí số (thập phân) là 0, 1, …, 9 có giá trị tương ứng là 0, 1, …, 9[14] còn hệ thống số cơ số 16 (*thập lục phân – hexadecimal*) dùng 16 kí số (thập lục phân) là 0, …, 9, A/a, B/b, C/c, D/d, E/e, F/f có giá trị tương ứng là 0, …, 9, 10, 11, 12, 13, 14, 15. Với hệ thập phân thì *chuỗi số*[15] 2017 mô tả cho số nguyên có giá trị là $2\times10^3 + 0\times10^2 + 1\times10^1 + 7\times10^0 = 2000 + 0 + 10 + 7 = 2017$ còn với hệ thập lục phân thì nó mô tả cho số nguyên có giá trị là $2\times16^3 + 0\times16^2 + 1\times16^1 + 7\times16^0 = 8192 + 0 + 16 + 7 = 8215$. Số nguyên có giá trị 2017 được mô tả trong hệ thập lục phân bằng chuỗi số 7E1 vì $7\times16^2 + 14\times16^1 + 1\times16^0 = 1792 + 224 + 1 = 2017$. Ta sẽ còn trở lại chủ đề quan trọng này sau, trước mắt, bạn cần nắm mang máng để có thể mô tả mã ASCII của kí tự theo cách truyền thống của C[16]. Một hằng số nguyên trong C có thể mô tả bằng cơ số 10 như thông thường hoặc bằng cơ số 16 với tiếp đầu ngữ `0x`. Như vậy trong mã C hằng số `2017` được hiểu là viết theo cơ số 10 (mô tả số nguyên 2017) còn hằng số `0x2017` được hiểu là viết theo cơ số 16 (mô tả số nguyên 8215). Đối với các kí tự in được mà gõ được từ bàn phím, C cho phép mô tả mã ASCII của kí tự đó bằng cách dùng *hằng kí tự* với cách viết: `'<kí tự>'`. Như vậy 3 lệnh gán sau đây là như nhau trong C:

```
ch = 65;
```

[11] Tưởng tượng ta lấy bút hết mực vẽ một nét ngang ta sẽ được một nét rỗng:)
[12] Xem Phụ lục A.4 để biết cách tùy chỉnh font chữ cửa sổ Console.
[13] Số viết theo cơ số 16 cũng hay được dùng để mô tả địa chỉ mà ta sẽ biết sau.
[14] Ta phải phân biệt được kí số với giá trị của nó. Kí số 0 là kí tự 0 (mã ASCII là 48) khác với giá trị 0 là số nguyên 0.
[15] Chuỗi số là dãy kí số (viết liên tiếp).
[16] Khi mô tả mã ASCII (chẳng hạn trên trang Wikipedia) người ta thường dùng số cơ số 16 (có hoặc không có số cơ số 10 tương ứng đi kèm).

```
ch = 0x41;
ch = 'A';
```

Cả 3 lệnh gán đều làm cho biến `ch` chứa mã ASCII của kí tự A. Dĩ nhiên ta nên dùng cách viết sau cùng vì nó rõ ràng và gợi nhớ nhất.

Đối với một số kí tự điều khiển thông dụng thì C cho phép dùng cách viết đặc biệt cho nó là dùng *chuỗi thoát* `\`<kí hiệu đặc biệt>. Chẳng hạn kí hiệu xuống dòng (LF, mã 10) có thể được viết là `\n`, kí hiệu kết thúc chuỗi (NULL, mã 0) là `\0`, kí hiệu tab (TAB, mã 9) là `\t`, kí hiệu âm cảnh báo[17] (BELL, mã 7) là `\a`, … Chuỗi thoát cũng được dùng để mô tả các kí tự in được đặc biệt của C, chẳng hạn ta không thể dùng hằng chuỗi `'''` để mô tả kí tự nháp đơn (')[18] mà phải dùng chuỗi thoát `\'` cho nó, nghĩa là dùng hằng chuỗi `'\''`. Tương tự ta cần dùng `\\`, `\"` cho các kí tự đặc biệt là \ và ". Hơn nữa C cho phép dùng chuỗi thoát `\x`<mã ASCII viết theo cơ số 16> để mô tả cho một kí tự bất kì. Như vậy kí tự xuống dòng (LF) có thể được mô tả bằng các cách: `10`, `0xA`, `'\n'`, `'\xA'`.

Như đã nói, kí tự là viên gạch để xây dựng nên dạng dữ liệu cực kì quan trọng là *chuỗi*. Chuỗi là dãy các kí tự. Hãy xem đoạn mã C sau:

```
char str[4];
str[0] = 'H';
str[1] = 'i';
str[2] = '!';
str[3] = '\0';
printf("%s", str);
```

với kết xuất:

```
Hi!
```

Đoạn mã trên như vậy xuất ra chuỗi `Hi!`. Dòng đầu tiên khai báo mảng[19] `str` chứa 4 kí tự liên tiếp, mà thực ra, như bạn đã biết là chứa mã ASCII của 4 kí tự. Dòng thứ 2 giúp đặt kí tự đầu tiên[20] cho chuỗi là kí tự H. Sau đó các kí tự i và ! được đặt tiếp vào. Sau cùng dòng 5 giúp đặt kí tự kết thúc chuỗi NULL. Như bạn đã biết đây là kí tự điều khiển, nó không được xuất ra mà nó được dùng để báo hiệu kết thúc chuỗi. Lệnh xuất với `%s` giúp xuất ra chuỗi trong `str` sau khi nó đã được đặt các kí tự mong muốn. Thật ra toàn bộ đoạn mã trên có thể được thay bằng một lệnh xuất chuỗi cố định mà bạn đã biết là:

```
printf("Hi!");
```

[17] Khi xuất kí tự BELL, máy sẽ kêu "bíp" nếu có loa trong.
[18] Kí tự nháy đơn (') là kí tự đặc biệt của C vì C dùng nó trong cách viết hằng kí tự.
[19] Ta sẽ tìm hiểu kĩ cái gọi là mảng sau, đừng lo lắng nếu bạn không biết nó là gì.
[20] Đầu tiên trong C nghĩa là thứ 0.

C cho phép mô tả một chuỗi cố định bằng cách dùng *hằng chuỗi* với cách viết "<chuỗi>", trong đó <chuỗi> là dãy các kí tự của chuỗi cần mô tả. Lưu ý là nếu chuỗi chứa các kí tự đặc biệt thì bạn cần phải dùng chuỗi thoát cho nó. Hơn nữa ta không mô tả kí tự kết thúc chuỗi vì C tự động đặt nó trong hằng chuỗi. Ta sẽ (phải) tìm hiểu kĩ về chuỗi sau. Trước mắt, bạn cần "cảm nhận" được cách chuỗi được tạo nên từ các kí tự và viết được mã C cho các xử lý "đơn giản" trên chuỗi. Chẳng hạn đoạn mã sau giúp đếm số từ trong một câu mà người dùng nhập.

Mã 1.7.1 – Đoạn mã đếm số từ trong một câu mà người dùng nhập

```
1  char sentence[1000];
2
3  printf("Enter a sentence: ");
4  gets(sentence);
5
6  int nwords = 1;
7  int length = strlen(sentence);
8  for(int i = 0; i < length; i++)
9      if(sentence[i] == ' ')
10         nwords = nwords + 1;
11
12 printf("Your sentence has %d word(s).", nwords);
```

Dòng 7 dùng hàm `strlen` để tính *chiều dài của một chuỗi*. Đó là số lượng kí tự trong chuỗi. Chẳng hạn chuỗi `Hi!` có chiều dài là 3. Hàm `strlen` là hàm của thư viện chuẩn, được khai báo trong `string.h` nên bạn cần phải có chỉ thị `#include` nó ở đầu chương trình[21]. Để đếm số lượng từ, mã trên dùng mẹo là đếm số lượng kí tự khoảng trắng (space). Hiển nhiên điều này chỉ đúng khi người dùng nhập câu có "định dạng phù hợp". Có lẽ hiện giờ, đoạn mã này hơi quá sức với bạn. Hãy xem lại Bài 1.2, Bài 1.4, Bài 1.5 và nghiền ngẫm để hiểu. *Hãy cảm nhận*!

Ngoài việc được dùng để mô tả/lưu trữ/truyền nhận dữ liệu chuỗi, các kí tự ASCII còn có những công dụng khác. Chẳng hạn, ta có thể chỉ dùng các kí tự ASCII mà vẫn tạo được bức hình sau đây:



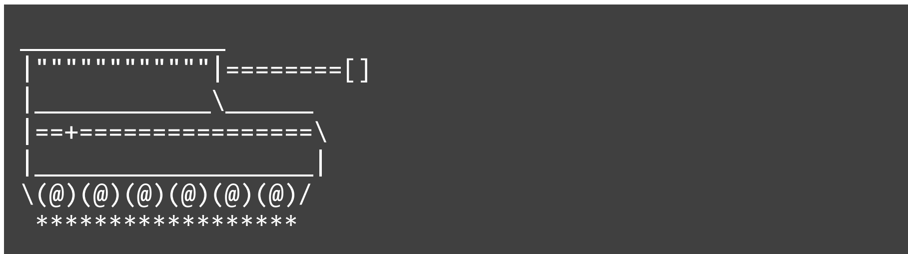
Chú mèo đáng yêu này được kết xuất từ đoạn mã C sau.

Mã 1.7.2 – Đoạn mã C vẽ hình chú mèo đáng yêu bằng các kí tự ASCII

[21] Trong `string.h` còn nhiều hàm xử lý chuỗi khác.

```
printf("  |\\_/|\n");
printf(" / @ @ \\\n");
printf("( > o < )\n");
printf(" `>>x<<\'\n");
printf(" /  O  \\\n");
```

Hình ảnh này cũng được lấy từ bài viết "ASCII art" trên trang Wikipedia[22]. Như tên gọi, các kí tự ASCII được dùng như một loại hình nghệ thuật đồ họa[23]. Như đoạn mã trên cho thấy, bức hình của ta thật ra là 5 dòng chuỗi kí tự ASCII. Lưu ý là ta đã dùng chuỗi thoát cho các kí tự đặc biệt: \n cho kí hiệu xuống dòng, \\ cho kí tự \ và \' cho kí tự '. Tương tự, bạn có thể viết mã C kết xuất ra hình chiếc xe tăng sau đây không?[24]



Để tăng phần kịch tích, ta sẽ nâng cấp các ảnh tĩnh ASCII thành các ảnh hoạt họa ASCII hay cao hơn nữa là các "video ASCII". Chương trình C sau đây sẽ "trình diễn" cảnh một chiếc trực thăng đang bay[25].

Mã 1.7.3 – Chương trình C minh họa trình diễn ảnh hoạt họa ASCII

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void delay(double dur)
{
    double s = clock(), t;
    do
    {
        t = (clock() - s)/CLOCKS_PER_SEC;
    }
    while(t < dur);
}

void frame1()
{
```

[22] https://en.wikipedia.org/wiki/ASCII_art
[23] Bên cạnh các loại hình đồ họa đúng nghĩa là raster và vector.
[24] Cũng từ https://en.wikipedia.org/wiki/ASCII_art
[25] Cũng từ https://en.wikipedia.org/wiki/ASCII_art

```
    printf("ROFL:ROFL:LOL:\n");
    printf("           A\n");
    printf("  L   /--------\n");
    printf("  O ===      []\\\n");
    printf("  L    \\        \\\n");
    printf("        \\--------]\n");
    printf("          I   I\n");
    printf("        ----------/\n");
}

void frame2()
{
    printf("         :LOL:ROFL:ROFL\n");
    printf("           A\n");
    printf("      /--------\n");
    printf(" LOL===      []\\\n");
    printf("       \\        \\\n");
    printf("        \\--------]\n");
    printf("          I   I\n");
    printf("        ----------/\n");
}

int main()
{
    for(int n = 20; n >= 0; n = n - 1)
    {
        system("cls");
        for(int i = 1; i <= n; i++)
            printf("\n");
        frame1();

        delay(0.3);

        system("cls");
        for(int i = 1; i <= n; i++)
            printf("\n");
        frame2();

        delay(0.3);
    }

    return 0;
}
```

Chạy chương trình này bạn sẽ thấy cảnh một chiếc trực thăng cất cánh. Thông cảm, hiện thời “video” của ta hơi bị “giựt”, ta sẽ học cách làm cho video trơn tru hơn sau. Để “trình diễn một đoạn video”, bí quyết của chương trình nằm ở chỗ: ta sẽ đều đặn kết xuất các khung hình khác nhau

sau mỗi khoảng thời gian ngắn[26]. Để làm được điều này ta cần làm được 3 việc:

(1) Xóa màn hình: khung hình cũ cần được xóa để vẽ lại khung hình mới.
(2) Tạm dừng chương trình một khoảng thời gian: một khung hình cần được hiển thị trong một khoảng thời gian trước khi bị xóa và thay bằng khung hình mới.
(3) Vẽ các khung hình: các khung hình, theo trình tự sẽ được hiển thị, tạo nên hình ảnh hoạt họa (hay video).

Mã 1.7.3 giải quyết 3 việc trên bằng cách:

(1) Gọi `system("cls")` để xóa màn hình (Dòng 43 và Dòng 50). Hàm `system` trong thư viện `stdlib.h` yêu cầu môi trường thực hiện một dịch vụ nào đó do chuỗi đối số xác định. Ở đây `cls` xác định dịch vụ xóa màn hình[27]. Dùng cách này để xóa màn hình không phải là cách hay vì nó gây giựt màn hình nhưng là cách đơn giản nhất.
(2) Gọi hàm `delay` với đối số thực xác định khoảng thời gian tạm dừng tính theo đơn vị là giây. Dòng 48 và Dòng 55 giúp tạm dừng và hiển thị mỗi khung hình trong 0.3 giây (khoảng 1/3 giây). Lưu ý, hàm `delay` là hàm do ta viết (từ Dòng 5 đến Dòng 13) mà trong đó ta đã dùng hàm `clock` (và hằng tượng trưng `CLOCKS_PER_SEC`) trong thư viện `time.h`. Ở đây bạn chưa cần hiểu hoạt động (bên trong) của hàm `delay` mà chỉ cần biết hàm `delay` giúp tạm dừng chương trình một khoảng thời gian (tính theo giây) để dùng nó là được[28].
(3) Hàm `frame1` và `frame2` giúp vẽ 2 khung hình khác nhau ứng với hai tình trạng khác nhau của trực thăng khi quay. Ảnh hoạt họa của ta ở đây khá nghèo nàn, chỉ có 2 khung hình khác nhau được luân phiên trình diễn lặp lại[29]. Để sinh động hơn, ta đã cho chiếc trực thăng không chỉ quay mà còn bay lên. Mẹo ở đây là ta thay đổi số dòng trống xuất ra trước khi kết xuất hình ảnh của máy bay. Ở đây, biến lặp n của vòng `for` ở Dòng 41 xác định số dòng trống này và bằng cách cho n giảm dần từ 20 đến 0 ta sẽ cho trực thăng bay từ dưới đất (cách đỉnh màn hình 20 dòng, n = 20) lên tới đỉnh màn hình (không có dòng trống nào, n = 0).

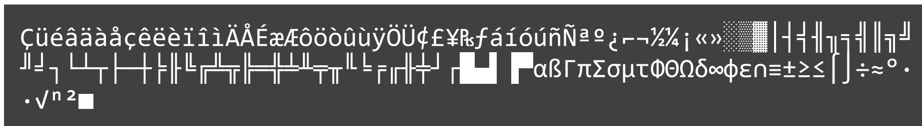
---

[26] Đây cũng là toàn bộ bí mật của nền công nghệ hoạt hình hay video mà thuật ngữ 24 fps có nghĩa là kết xuất 24 khung hình trong 1 giây (24 frames per second).
[27] Các dịch vụ và chuỗi xác định nó tùy thuộc vào môi trường, ở đây là trên môi trường Windows, các môi trường khác sẽ có chuỗi tương ứng khác.
[28] Nghĩa là ta dùng hàm `delay` như một hộp đen.
[29] Bạn hãy xem ảnh hoạt họa này trên https://en.wikipedia.org/wiki/ASCII_art để hình dung nhé.

Thực sự thì Mã 1.7.3 quá sức với bạn lúc này[30]. Không cần cảm nhận:), bạn chỉ cần hiểu mang máng các chi tiết kĩ thuật phức tạp. Thứ mà bạn cần tập trung và hiểu rõ ở đây là kết xuất các kí tự ASCII (hàm `frame1` và `frame2`), những thứ khác bạn cứ "bắt chước – chỉnh sửa – phân tích – phát triển" hay nói gọn "làm đại" là được. Bạn có biết cách làm cho trực thăng bay nhanh/chậm hay cao/thấp hơn không?

Mã ASCII chỉ mã cho 128 kí hiệu, nghĩa là chỉ cần 7 bít[31]. Tuy nhiên trên máy tính (computer) và các hệ máy tương tự, do tính chẵn byte mà máy sẽ dùng trọn vẹn một byte để mã các kí tự. Vì 1 byte gồm 8 bít nên mã được cho 2^8 = 256 kí tự với mã số từ 0 đến 255. Như vậy mã ASCII, một cách tự nhiên, sẽ được mở rộng để mã thêm cho 128 kí hiệu nữa với mã số từ 128 đến 255 ngoài 128 kí hiệu ASCII gốc với mã số từ 0 đến 127. Những bảng mã như vậy được gọi là bảng mã *ASCII mở rộng*. Khác với 128 kí tự đầu của mã ASCII, 128 kí tự sau đó không có sự thống nhất cao và thường được dùng cục bộ (hay riêng tư) cho các quốc gia, vùng miền, ngôn ngữ, hệ điều hành, hệ máy, hệ thống, … khác nhau cho các mục đích khác nhau. Một bảng mã ASCII mở rộng được dùng khá phổ biến là bảng mã của hệ máy IBM PC, hệ điều hành DOS (và các hậu duệ) là *bảng mã 437*. Đoạn mã sau sẽ giúp xuất ra 128 kí tự mở rộng (mã từ 128 đến 255). Lưu ý là phải dùng kiểu `int` cho biến lặp `ch` chứ không phải kiểu `char` vì một lý do hiển nhiên mà ta sẽ biết sau. Cũng lưu ý là cửa sổ Console phải được thiết lập phù hợp để hiển thị các kí tự của bảng mã 437[32].

```
for(int ch = 128; ch <= 255; ch++)
    printf("%c", ch);
```



Bạn cũng có thể dùng đoạn mã sau để xuất bảng mã đẹp hơn và có mã số theo cơ số 16 đi kèm:

```
for(int ch = 128; ch <= 255; ch++)
{
    printf("%4x:%2c", ch, ch);
    if((ch - 127) % 5 == 0)
        printf("\n");
}
```

[30] Cũng như phần mở rộng của các bài học, bài kết thúc của mỗi tầng là bài tổng hợp, mở rộng, nâng cao, … với khối lượng khá lớn và hơi quá sức. Bạn sẽ (chỉ) thông suốt nó khi luyện qua các tầng cao hơn.

[31] 128 = 2^7.

[32] Xem Phụ lục A.4 để biết cách cấu hình cửa sổ Console.

Bạn hãy tham khảo bài viết “Code page 437” trên trang Wikipedia[33] để biết rõ hơn. Ở trển cũng trình bày bảng mã rất đẹp.

Bảng mã 437 có một số kí tự nguyên âm có dấu dùng ở một số nước châu Âu mà trong đó có vài kí tự tiếng Việt như é, â, à, ê, … Bảng mã này cũng có những kí hiệu toán thông dụng và đặc biệt là có nhiều kí hiệu đồ họa giúp vẽ các khối, đoạn, … Chẳng hạn đoạn mã C sau đây sẽ xuất ra lời chào tiếng Việt được đóng khung trang trọng:

Mã 1.7.4 – Đoạn mã C xuất lời chào tiếng Việt bằng các kí tự mở rộng

```
printf("\xda\xc4\xc4\xc4\xc4\xc4\xc4\xbf\n");
printf("\xb3 Ch\x85o \xb3\n");
printf("\xc0\xc4\xc4\xc4\xc4\xc4\xc4\xd9\n");
```

Ở đây do các kí hiệu mở rộng này không gõ được từ bàn phím nên ta phải dùng chuỗi thoát \x<mã số viết theo cơ số 16> để mô tả cho kí hiệu tương ứng. Chẳng hạn kí hiệu à có mã là 133, viết theo cơ số 16 là 85[34] hay kí hiệu đồ họa góc trên trái có mã là 218, viết theo cơ số 16 là da, …

MỞ RỘNG 1.7 – Mã Unicode và bộ kí hiệu tiếng Việt

Ngoài tính phổ biến, mã ASCII còn có một ưu điểm nữa là rất tiết kiệm: chỉ tốn 7 bít để biểu diễn/lưu trữ/truyền một kí tự ASCII vì chỉ cần 7 bít để biểu diễn các số nguyên từ 0 đến 127. Tuy nhiên đây cũng chính là hạn chế của nó: 128 kí tự ASCII là đủ cho bộ kí hiệu Mỹ nhưng không đủ bên ngoài nước Mỹ:) Chẳng hạn 128 kí tự ASCII và cả 128 kí tự mở rộng cũng không đủ cho tất cả các kí hiệu tiếng Việt[35]. Trong thời đại toàn cầu hóa ngày nay, thực sự chỉ có một bộ kí hiệu: bộ kí hiệu dùng chung cho toàn thể nhân loại. Bộ kí hiệu này gồm các kí hiệu ASCII, các kí hiệu tiếng Việt, tiếng Nhật, tiếng Nga, …, các kí hiệu Toán, Lý, … . Với bộ kí hiệu khủng này, mã ASCII chỉ là một phần nhỏ bé do đó ta cần một bảng mã khủng hơn, *mã Unicode*, mà rất may, nó cũng đang được dùng phổ biến và thống nhất toàn cầu.

Giống mã ASCII, mã Unicode cũng dùng các số nguyên không âm đầu tiên để mã cho các kí tự. Theo truyền thống, *mã số* (*code point*) này được viết theo cơ số 16 với tiếp đầu ngữ là U+. Chẳng hạn kí tự A (hoa) có mã là 65[36] mà viết theo cơ số 16 là 41 nên được gọi là kí hiệu Unicode U+41.

---

[33] https://en.wikipedia.org/wiki/Code_page_437

[34] Tới đây chắc bạn đã hiểu chuỗi xuất bí hiểm trong phần Mở rộng 1.1.

[35] Đã có nhiều bảng mã ASCII mở rộng được Việt hóa cho bộ kí tự tiếng Việt trong thời kì quá độ lên Unicode:) Chúng đều thành di sản văn hóa máy tính Việt Nam:)

[36] 128 code point đầu tiên (0, 1, …, 127) của mã Unicode giống mã ASCII của 128 kí tự ASCII.

Bạn hãy tham khảo bài viết Unicode trên trang Wikipedia[37] để biết sơ lược, ta sẽ tìm hiểu chi tiết hơn sau. Hiển nhiên là toàn bộ kí hiệu tiếng Việt đều có trong bảng mã Unicode. Chẳng hạn kí tự Á là U+C1, À là U+C0, Ả là U+1EA2, Ã là U+C3, Ạ là U+1EA0, … Có rất nhiều công cụ để bạn có thể xem/tra mã Unicode. Chẳng hạn tôi đã dùng công cụ “Unicode code converter”[38] để tra mã các kí tự tiếng Việt trên[39].

Vì mã Unicode mã hóa cho khoảng hơn 1 triệu kí tự nên cần 21 bít[40]. Tuy nhiên theo tính “chẵn byte” thì máy sẽ dùng 4 byte (32 bít) cho 1 kí hiệu. Do đó đơn giản nhất thì mỗi kí tự sẽ là một số nguyên 4 byte trên máy (như kiểu `int` của C). Cách triển khai Unicode đơn giản này được gọi là UCS-4 hay UTF-32[41]. Nó có ưu điểm là đơn giản trong cách mã và xử lý nhưng có nhược điểm là tốn kém nhiều không gian lưu trữ hay truyền/nhận chuỗi kí tự. Để khắc phục hạn chế này, Unicode dùng một số *lược đồ mã* (encoding) tiết kiệm cho các code point Unicode mà phổ biến nhất là UTF-8[42]. UTF-8 dùng 1 đến 4 byte để mã các code point Unicode mà nói chung những kí tự hay gặp nhất sẽ được mã bằng ít byte (các kí tự ASCII được mã bằng 1 byte) còn các kí tự ít dùng sẽ được mã bằng nhiều byte. Chính nhờ sự tiết kiệm này mà UTF-8 là cách mà Unicode được triển khai phổ biển. Bù lại, phương pháp mã này gây khó khăn cho việc xử lý.

Ta khép lại bài này và cũng là Tầng khởi động, làm quen này bằng một chương trình Hello phiên bản tiếng Việt đúng nghĩa. Ở phần Mở rộng 1.1 tôi đã viết chương trình đưa ra lời chào đến mình bằng tiếng Việt “Chào Hoàng”. Ở đây tôi sẽ đưa ra lời chào đầy đủ hơn bằng chương trình C bên dưới.

Mã 1.7.5 – Chương trình Hello phiên bản tiếng Việt đúng nghĩa

```
#include <stdio.h>
#include <windows.h>

int main()
{
    SetConsoleOutputCP(65001);

    printf("Ch\xC3\xA0o V\xC5\xA9");
```

[37] https://en.wikipedia.org/wiki/Unicode
[38] https://r12a.github.io/apps/conversion
[39] Công cụ này không phải tôi viết, cũng không phải bạn tôi viết, không có ý quảng bá gì nha:) Đơn giản là search Google “Unicode tool” thì nó lòi ra đầu tiên. Bạn có thể dùng một công cụ khác (trong một nghìn lẻ một công cụ như vậy).
[40] $2^{21} = 2097152$.
[41] Tham khảo https://en.wikipedia.org/wiki/UTF-32.
[42] Tham khảo https://en.wikipedia.org/wiki/UTF-8.

```
 9     printf(" Qu\xE1\xBB\x91\c Ho\xC3\xA0ng");
10 }
```

Chương trình xuất ra lời chào “Chào Vũ Quốc Hoàng”[43]. Như đã thấy kí tự à có trong bảng mã ASCII mở rộng 437 nhưng kí tự ũ và ố thì không. Ở đây ta sẽ dùng bảng mã Unicode với lược đồ mã UTF-8. Cần lưu ý 3 điều:

(1) Ta cần đặt lược đồ mã cho cửa sổ Console là UTF-8. Điều này tùy thuộc vào môi trường thực thi (người quản lý cửa sổ Console). Trên hệ điều hành Windows ta có thể dùng hàm `SetConsoleOutputCP` trong `windows.h` với đối số là mã số của bảng mã. Như bạn thấy ở Dòng 6, 65001 là mã số của lược đồ mã UTF-8[44]. Lưu ý là `SetConsoleOutputCP` không phải là hàm trong thư viện chuẩn của C mà là hàm trong thư viện của môi trường (ở đây là Windows). Khác với những hàm trong thư viện chuẩn, những hàm của môi trường chỉ có thể dùng trong các chương trình thực thi trên môi trường đó. Chương trình trên như vậy chỉ có thể chạy trên Windows. Để chạy trên các môi trường khác, ta cần dùng hàm tương ứng của môi trường đó.

(2) Ta cần biết dãy byte UTF-8 của kí tự tương ứng. Như đã nói, các kí tự ASCII chỉ gồm 1 byte và giống với mã ASCII nên có thể dùng như thông thường. Các kí tự à, ũ, ố không phải là các kí tự ASCII. Cũng dùng công cụ “Unicode code converter” ta tra được à gồm 2 byte viết theo cơ số 16 là C3 A0, ũ gồm 2 byte là C5 A9 và ố gồm 3 byte là E1 BB 91. Dùng chuỗi thoát \x… cho các byte đặc biệt trên ta có chuỗi xuất ra như ở Dòng 8 và Dòng 9. Hiển nhiên là bạn có thể gom hai dòng xuất đó thành một.

(3) Để kết quả hiển thị đúng, nhất là các kí tự à, ũ, ố thì ta cần thiết lập font chữ phù hợp cho cửa sổ Console. Khi một font chữ hiển thị được/đúng hình dạng của một kí hiệu, ta nói font chữ có *hỗ trợ* kí hiệu đó. Hầu hết các font chữ hỗ trợ đầy đủ các kí tự ASCII. Có nhiều font chữ hỗ trợ nhiều kí tự Unicode mà ta gọi là các *font Unicode*. Thuật ngữ này không có nghĩa là font chữ đó hỗ trợ tất cả các kí hiệu Unicode mà là hỗ trợ nhiều kí tự cơ bản. Rất may là các kí tự tiếng Việt cũng thường nằm trong số các kí tự đó. Cũng có những font chữ Unicode được thiết kế riêng cho tiếng Việt hay tiếng nào đó, nghĩa là nó chủ yếu chỉ hỗ trợ các kí hiệu tương ứng

[43] It’s me.

[44] Mã số các bảng mã khác trên Windows xem tại https://msdn.microsoft.com/en-us/libray/windows/desktop/dd317756(v=vs.85).asps

> đó. Với cửa sổ Console trên Windows bạn có thể chọn font chữ Unicode phổ biến là Consolas[45].
>
> Bạn có thể xuất ra lời chào đến mình như tôi đã làm không? *Ôi, ta đi xa quá!*:)

BÀI TẬP

Bt 1.7.1 Bạn đã biết đến ASCII Art, hãy làm lại Bài tập 1.1.4 với kết xuất đẹp hơn. (*Gợi ý*: có rất nhiều mẫu ASCII Art mà bạn có thể tham khảo, chẳng hạn tham khảo kho ASCII Art tại https://www.ascii-code.com/ascii-art/.)

Bt 1.7.2 Viết chương trình để kết xuất ra chiếc xe tăng như trong bài.

Bt 1.7.3 Viết chương trình trình diễn cảnh chiếc xe tăng chạy tới. (*Gợi ý*: tương tự Mã 1.7.3 và xuất các kí tự trắng ở bên trái chiếc xe tăng.)

Bt 1.7.4 Viết chương trình dùng các kí tự đồ họa trong bảng mã 437 để kết xuất giống hình tại mục "Block or High ASCII style"[46].

Bt 1.7.5 Tìm hiểu một bảng mã ASCII mở rộng khá phổ biến nữa là Windows-1252. Viết chương trình hiển thị bảng mã này trên máy.

Bt 1.7.6 Làm lại Bài tập 1.1.2 với kết xuất tiếng Việt có dấu. (*Gợi ý*: dùng bảng mã Unicode với lược đồ mã UTF-8 tương tự Mã 1.7.5.)

---

[45] Xem Phụ lục A.4 để biết cách cấu hình.

[46] https://en.wikipedia.org/wiki/ASCII_art#/media/File%3AAa_example3.png

BÀI 2.1

Ta đã biết mã C là dãy các từ (token). Các từ này ghép lại theo qui tắc để tạo thành các cấu trúc cú pháp cao hơn. Trong C, *biểu thức* là cấu trúc cú pháp cơ bản nhất và quan trọng nhất, nó là thành tố dùng để xây dựng nên cấu trúc cú pháp quan trọng khác là lệnh. Nếu xem lệnh như là câu (mà ta hay gọi là câu lệnh) thì biểu thức có thể xem là cụm từ. Ta có nhiều loại cụm từ trong tiếng Việt tùy theo nội dung mà cụm từ đó mô tả: cụm danh từ mô tả một đối tượng, cụm động từ mô tả một hành động hay trạng thái, cụm tính từ mô tả một tính chất, … Ta gọi nội dung mà cụm từ mô tả là *ngữ nghĩa* của cụm từ đó. *Ngữ nghĩa của một biểu thức trong C là một giá trị* (một dữ liệu cụ thể của một dạng dữ liệu nào đó) và việc tính giá trị của biểu thức còn được gọi là *lượng giá* biểu thức.

Về mặt cấu tạo, ta có có hai loại biểu thức:

- *Biểu thức đơn*: là biểu thức chỉ gồm một từ, có thể là:
  - *Hằng số*: hằng số nguyên (như `3`), hằng số thực (như `3.1416` hay `1e-12`[1]), hằng kí tự (như `'3'`). Khi đó giá trị của biểu thức chính là giá trị của hằng số đó.
  - *Biến*: như biến nguyên, biến thực, biến kí tự, biến chuỗi, … có giá trị là dữ liệu đang chứa trong biến đó.
- *Biểu thức phức*: là biểu thức gồm nhiều từ, tạo nên bởi các biểu thức nhỏ hơn và các toán tử. Các biểu thức con nhỏ hơn đó được gọi là *toán hạng*. Các *toán tử* kí hiệu cho các *phép toán*, là các thao tác tính toán trên các giá trị của các toán hạng. Số lượng các toán hạng cần cho phép toán mà toán tử kí hiệu được gọi là *số ngôi* của toán tử đó. Ví dụ biểu thức `12 + 3` là biểu thức phức với hai toán hạng là `12`, `3`. Toán tử `+` kí hiệu cho phép cộng, phép toán này đòi hỏi hai toán hạng (cộng cái gì đó với cái gì đó) nên `+` là toán tử hai ngôi. Toán hạng `12`, là biểu thức đơn, có giá trị là 12. Toán hạng `3` có giá trị là 3. Biểu thức `12 + 3` có giá trị là 15, chính là kết quả của thao tác cộng giá trị 12 với giá trị 3. Đa số các toán tử trong C là hai ngôi, một vài toán tử là một ngôi, có duy nhất một toán tử ba ngôi và không có toán tử nào nhiều ngôi hơn.

Điều làm cho các biểu thức phức thực sự phức tạp là các biểu thức con của nó có thể là biểu thức đơn nhưng cũng có thể là biểu thức phức khác. Ví

---

[1] Đó là cách viết hằng số thực theo dạng khoa học của C cho số $1\times10^{-12} = 10^{-12}$. Xem Phụ lục A.5 để biết cách viết đúng các hằng số.

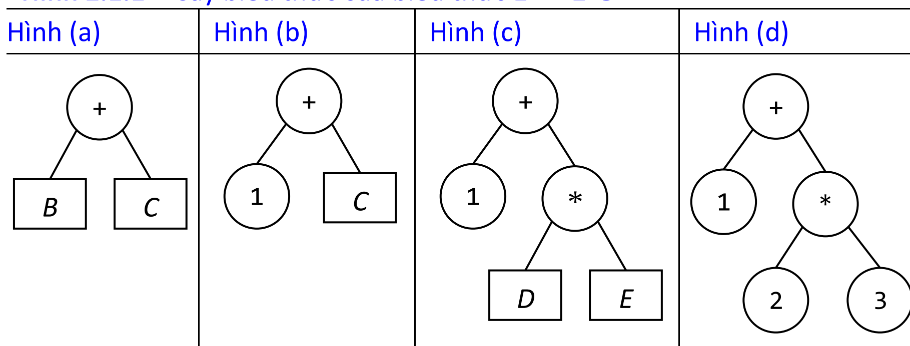
dụ: 1 + 2*3 có hai toán hạng là 1 và 2*3 trong đó 1 là biểu thức đơn nhưng 2*3 là một biểu thức phức. Số mức phức như vậy là không giới hạn.

Khi biểu thức phức có nhiều toán tử thì C dùng 3 qui tắc sau đây để xác định thứ tự thực hiện các phép toán:

- Biểu thức trong ngoặc được tính toán trước.
- Phép toán ứng với toán tử có *độ ưu tiên* cao hơn thì được thực hiện trước: chẳng hạn, "nhân chia trước cộng trừ sau" vì toán tử *, / có độ ưu tiên cao hơn toán tử +, –.
- Các phép toán cùng độ ưu tiên thì thực hiện theo *tính kết hợp* của phép toán: chẳng hạn, "cùng nhân chia hoặc cộng trừ thì từ trái qua phải" vì các toán tử *, /, +, – đều có tính kết hợp trái. Đa số các toán tử trong C có tính kết hợp trái như vậy nhưng cũng có một số toán tử có tính kết hợp phải, khi đó chúng ta sẽ tính từ phải qua[2].

Phân tích biểu thức là cho thấy cấu tạo của biểu thức: gồm các toán tử nào, mấy ngôi, các toán hạng nào và thứ tự thực hiện các phép toán. Chẳng hạn với biểu thức *A* là 1 + 2*3, thì *A* là biểu thức phức có dạng *B* + *C* với toán tử hai ngôi +. Toán hạng *B* là biểu thức đơn 1. Toán hạng *C* là biểu thức phức 2*3 có dạng *D***E* với toán tử hai ngôi *. Toán hạng *D* là biểu thức đơn 2 còn toán hạng *E* là biểu thức đơn 3. Phép toán nhân được thực hiện trước phép toán cộng do * có độ ưu tiên cao hơn +. Quá trình phân tích ở trên có thể được mô tả gọn gàng, cô đọng qua một sơ đồ gọi là *cây biểu thức*. Biểu thức *A* có dạng *B* + *C* được mô tả bằng cây có *nút* là toán tử + và hai *nhánh* trái phải tương ứng là *B*, *C* như Hình (a) sau. Vì *B*, *C* chưa được phân tích nên nó có dạng nút vuông, còn nút tròn là nút đã phân tích xong. Phân tích *B*: *B* là biểu thức đơn 1, ta dùng nút tròn 1 cho nó được Hình (b). Phân tích *C*: *C* có dạng *D**E nên được Hình (c). Vì *D* là biểu thức đơn 2, *E* là biểu thức đơn 3 nên được Hình (d). Tới đây ta không còn nút vuông nữa, nghĩa là ta đã phân tích xong biểu thức *A* và Hình (d) gọi là cây biểu thức của biểu thức *A*.

Hình 2.1.1 – Cây biểu thức của biểu thức 1 + 2*3



[2] Xem Phụ lục A.6 để thấy danh sách các toán tử của C với độ ưu tiên và tính kết hợp của chúng.

Ta nhận thấy cây biểu thức mô tả đầy đủ và cô đọng cấu tạo của biểu thức. Nút có nhánh được gọi là *nút nội*, đó chính là các nút toán tử với số nhánh chính là số ngôi của toán tử. Nút không có nhánh được gọi là *nút lá*, đó chính là các nút ứng với biểu thức đơn (là hằng hoặc biến). Cây biểu thức cũng cho thấy thứ tự lượng giá các biểu thức: các *nút con* (ứng với biểu thức con) phải được lượng giá trước *nút cha*. Chẳng hạn trong biểu thức *A* thì phép nhân thực hiện trước phép cộng do nút * là con của nút + (Hình (d) trên). Với những biểu thức không có hiệu ứng lề[3] thì ta có thể tính giá trị của biểu thức bằng cách tính giá trị cho từng nút trong cây từ *mức* cao xuống mức thấp (từ dưới lên trên), trong từng mức thì tính cho từng nút từ trái qua phải. Nút lá là số thì có giá trị chính là số đó, còn nút lá là biến thì giá trị là giá trị đang chứa trong biến đó. Nút nội có giá trị là kết quả của phép toán kí hiệu bởi toán tử trong nút đó với các toán hạng là giá trị của các nút con. Ví dụ: ta tính giá trị của biểu thức *A* từ cây biểu thức của A (Hình (a) dưới) bắt đầu từ mức cao nhất là mức 3 (hàng dưới cùng), từ trái qua phải, giá trị của nút 2 là 2 và giá trị của nút 3 là 3 (Hình (b)). Sau đó lên mức 2 thì từ trái qua: nút lá 1 có giá trị là 1, nút nội * có giá trị là kết quả của nhân 2 với 3 là 6. Lên mức 1 thì nút nội + có giá trị là kết quả của cộng 1 với 6 là 7 (Hình (c)). Nhận xét rằng mọi cây biểu thức đều chỉ có một nút ở mức 1, gọi là *nút gốc* của cây. *Giá trị của nút gốc chính là giá trị của biểu thức tương ứng mà cây biểu diễn*. Như vậy biểu thức *A* (1 + 2*3) có giá trị là 7.

Hình 2.1.2 – Tính giá trị từ cây biểu thức của biểu thức 1 + 2*3

	Hình (a)	Hình (b)	Hình (c)
Mức 1	+	+	+ (7)
Mức 2	1, *	1, *	1 (1), * (6)
Mức 3	2, 3	2 (2), 3 (3)	2 (2), 3 (3)

Tương tự ta có cây biểu thức và giá trị tương ứng của biểu thức *A* là 1 – 2 – 3 (Hình (a) dưới), *B* là 1 – (2 – 3) (Hình (b)) và *C* là –x * 4 (giả sử x là biến nguyên đang chứa giá trị 5) (Hình (c)).

Ở biểu thức *A* thì toán tử – hai ngôi (kí hiệu phép trừ) có tính kết hợp trái nên phép trừ thứ nhất được thực hiện trước phép trừ thứ hai (nghĩa là thực hiện 1 – 2 trước), do đó kết quả của *A* là -4. Ở biểu thức *B* thì biểu thức con 2 – 3 trong cặp ngoặc tròn được thực hiện trước nên kết quả của

[3] Ta sẽ tìm hiểu biểu thức có hiệu ứng lề trong bài sau.

biểu thức *B* là 2. Ở biểu thức *C* thì toán tử - kí hiệu cho phép toán lấy đối (âm thành dương, dương thành âm) nên là toán tử một ngôi[4]. Vì toán tử này là một ngôi nên nút tương ứng cho nó (nút - ở mức 2) chỉ có một nhánh. Cũng lưu ý là toán tử - một ngôi có độ ưu tiên cao hơn toán tử * nên được thực hiện trước. Nút lá x là biến và có giá trị là 5, chính là giá trị đang chứa trong biến x. Biểu thức C, như vậy, có kết quả là -20.

Hình 2.1.3 – Cây biểu thức và giá trị của các biểu thức

	Hình (a): 1–2–3	Hình (b): 1–(2–3)	Hình (c): -x*4
Mức 1	-4 –	2 –	* -20
Mức 2	-1 – 3 3	1 1 – -1	- -5 4 4
Mức 3	1 1 2 2	2 2 3 3	x 5

Cũng lưu ý là cách phân tích biểu thức bằng cây biểu thức là cách mà các compiler hay dùng, tuy nhiên việc có xây dựng cây tường minh hay không và cách duyệt qua cây để tính giá trị biểu thức thì có thể khác nhau giữa các compiler. Cách duyệt cây để tính giá trị biểu thức ở trên (tính từ dưới lên, trái qua phải) là cách dễ hình dung với ta nhưng không phù hợp với máy nên thường không được các compiler dùng[5].

Ngoài ra ta cũng có một cách phân tích và tính giá trị biểu thức đơn giản "*bằng mắt*" như minh họa trong ví dụ sau. Chẳng hạn ta cần tính giá trị biểu thức: -(x + 2)*4 – 5 với biến x đang chứa giá trị 1. Đầu tiên ta thấy cần thực hiện phép + trước do qui tắc làm trong ngoặc trước: -(x + 2)*4 – 5. Thực hiện phép toán này ta được giá trị là 3 (do x đang chứa giá trị 1 và 1 + 2 được 3). Do đó ta được biểu thức mới là -3*4 – 5. Tiếp theo thì ta cần thực hiện phép toán lấy đối (do toán tử - một ngôi có độ ưu tiên cao hơn * và – hai ngôi): -3*4 – 5. Vì đối của số nguyên dương 3 là số nguyên âm 3 nên ta được biểu thức mới là: -3*4 – 5. Lưu ý dấu - nhỏ trước số 3 là để kí hiệu rằng -3 là số âm 3 (ta đọc -3 như một tổng thể là số nguyên âm 3, khác với -3 là biểu thức phức, là đối (toán tử - một ngôi) của biểu thức đơn là số nguyên dương 3)[6]. Tiếp theo vì toán tử * có độ ưu tiên hơn toán tử – (hai

[4] Phân biệt với toán tử – 2 ngôi kí hiệu cho phép trừ như ở biểu thức *A*.

[5] Ta sẽ tìm hiểu sâu hơn sau.

[6] Đây là hiện tượng "*lạm dụng kí hiệu*": cùng kí hiệu - được kí hiệu cho phép toán đối (- một ngôi), phép toán trừ (- hai ngôi) và số âm. Ta cần dựa vào ngữ cảnh để xác định ngữ nghĩa của nó (tức là nó kí hiệu cho cái gì trong 3 cái trên).

ngôi) nên ta phải thực hiện phép nhân trước: -3*4 – 5 được -12 – 5 được -17[7].

Ta đã thấy rằng ngữ nghĩa (hay nội dung) của biểu thức là giá trị. *Bất kì chỗ nào trong chương trình cần một giá trị thì ta có thể mô tả nó bằng một biểu thức* và biểu thức sẽ phải được tính giá trị rồi chương trình mới dùng giá trị đó thực hiện tiếp. Giống như cụm từ trong tiếng Việt được phân loại dựa trên ngữ nghĩa của nó như cụm danh từ, cụm động từ, cụm tính từ, … Biểu thức trong C cũng được phân loại dựa trên giá trị của nó. Theo đó, ta có *biểu thức gốc số* và *biểu thức phi số*[8]. Biểu thức gốc số[9] là biểu thức có giá trị là số nguyên hoặc thực. Tùy theo mục đích sử dụng giá trị số đó mà ta có các loại biểu thức sau:

- *Biểu thức số học*: giá trị số được sử dụng theo cách thông thường.
- *Biểu thức kí tự*: giá trị số (nguyên) được dùng để xác định một kí tự, nói cách khác là *mã số của kí tự*[10].
- *Biểu thức luận lý*: giá trị số được dùng để xác định *chân trị* Đúng/Sai (True/False) hay Có/Không. Qui tắc của C là: giá trị số là 0 (nguyên) hay 0.0 (thực) là Sai và khác 0 (hay khác 0.0) là Đúng.
- *Biểu thức địa chỉ*: giá trị số (nguyên không âm) được dùng như là *địa chỉ* để xác định một vùng nhớ, như địa chỉ của biến xác định vùng nhớ chứa giá trị của biến.

Biểu thức số học sử dụng các giá trị số cho các tính toán thông thường với các toán tử số học: + một ngôi, - một ngôi (lấy đối); các toán tử hai ngôi: +, -, *, /, %[11]. Cũng nên biết rằng việc tính toán trên số thực là khác với số nguyên[12] nên thực ra cùng một kí hiệu toán tử nhưng có thể kí hiệu cho 2 phép toán tương ứng khác nhau trên số thực hay số nguyên[13]. Ví dụ: trong biểu thức `1 + 2` thì toán tử + kí hiệu cho phép toán cộng số nguyên nhưng trong biểu thức `0.5 + 2.5` thì toán tử + kí hiệu cho phép toán cộng số thực. Sự nhập nhằng này được giải quyết bằng *kiểu của các toán hạng* (là nguyên hay thực). Cũng lưu ý trong biểu thức `1 + 2.5` thì do một toán hạng là số nguyên, một toán hạng là số thực nên không rõ ràng là sẽ dùng phép toán cộng nguyên hay cộng thực. Trong trường hợp như vậy, C sẽ tự động chuyển giá trị nguyên 1 sang giá trị thực 1.0 rồi thực hiện phép cộng thực với 2.5. Trong trường hợp bạn muốn tự mình chỉ định việc chuyển kiểu thì

---

[7] Lần nữa, đây là số nguyên âm 17.

[8] Thực ra còn loại nữa là *biểu thức không có giá trị* hay *biểu thức void* mà ta sẽ tìm hiểu sau.

[9] Biểu thức phi số sẽ được tìm hiểu sau.

[10] Là *mã ASCII* như đã thấy trong Bài 1.7.

[11] Một số ngôn ngữ lập trình dùng toán tử ^ cho phép toán lấy mũ nhưng C không có toán tử này. (Đúng hơn, C dùng toán tử ^ cho một phép toán khác.)

[12] Tính toán trên số thực thì phức tạp hơn trên số nguyên nhiều.

[13] Chính là hiện tượng lạm dụng kí hiệu mà ta đã nói.

có thể dùng *toán tử chuyển kiểu* một ngôi: `(int)` để chuyển giá trị thực thành nguyên (bằng việc cắt bỏ phần lẻ) hay `(double)` để chuyển giá trị nguyên thành thực. Ví dụ: trong biểu thức `1 + (int)2.5` thì giá trị thực 2.5 được chuyển thành giá trị nguyên 2 sau đó được cộng nguyên với giá trị nguyên 1, trong biểu thức `(double)1/2` thì giá trị nguyên 1 được chuyển thành giá trị thực 1.0 thành `1.0/2`, giá trị 2 nguyên được chuyển tự động thành 2.0 thực thành `1.0/2.0` được chia theo kiểu thực được 0.5[14]. Cũng lưu ý là toán tử chia dư `%` không được dùng với số thực. C sẽ báo lỗi ngữ nghĩa với biểu thức: `1.0 % 2.0`. Cách thực hiện phép chia nguyên (`/` và `%`) cũng hơi rắc rối khi làm việc với số nguyên âm (xem lại Bài 1.2 để nắm rõ).

Biểu thức luận lý là biểu thức mà giá trị của nó là *Đúng* hoặc *Sai* và thường được dùng để mô tả một điều kiện hay một kết quả kiểm tra nào đó. Trong trường hợp tính giá trị luận lý ra số thì C sẽ dùng số nguyên 1 cho Đúng và số nguyên 0 cho Sai. Các *toán tử quan hệ* (hay so sánh) hai ngôi là: `>`, `>=` (cho so sánh ≥), `<`, `<=` (cho so sánh ≤), `==` (cho so sánh =) và `!=` (cho so sánh ≠) nhận các toán hạng là biểu thức số học và cho kết quả Đúng hoặc Sai, nghĩa là 1 hoặc 0 tùy kết quả so sánh. Chẳng hạn `10*0 < 1` có kết quả là 1 (Đúng) và `(int)2.9 > 2` có kết quả là 0 (Sai). Cũng lưu ý là để so sánh bằng chúng ta dùng toán tử `==`[15] còn toán tử `=` là toán tử gán.

Biểu thức luận lý cũng có thể được tạo nên từ các biểu thức luận lý nhỏ hơn và các *toán tử logic* để mô tả các điều kiện phức tạp. Các toán tử `!`, `||`, `&&` mô tả cho các phép toán logic "phủ định", "hoặc", "và" tương ứng và có ý nghĩa như bảng sau.

Bảng 2.1.1 – Bảng ngữ nghĩa của các toán tử logic

A	*B*	!*A*	*A* || *B*	*A* && *B*
Sai	Sai	Đúng	Sai	Sai
Sai	Đúng	Đúng	Đúng	Sai
Đúng	Sai	Sai	Đúng	Sai
Đúng	Đúng	Sai	Đúng	Đúng

Để dễ nhớ: phủ định cho kết quả ngược lại, *A* && *B* chỉ Đúng khi cả *A* và *B* đều Đúng, *A* || *B* chỉ Sai khi cả *A* và *B* đều Sai. Lưu ý phép toán hoặc (`||`), như vậy, mang nghĩa *bao hàm* (*A* || *B* Đúng trong cả trường hợp *A* và *B* đều Đúng)[16].

Trong trường hợp nhận giá trị luận lý thì bất kì giá trị số nào khác 0 (hay khác 0.0) được xem là Đúng và là 0 (hay 0.0) được xem là Sai. Chẳng hạn `(2`

---

[14] Toán tử ép kiểu có độ ưu tiên cao hơn +, -, *, /, %.

[15] Hai dấu = viết liên tiếp.

[16] Một vài ngôn ngữ có phép toán logic *hoặc loại trừ* (XOR), thường được kí hiệu là ^, cho kết quả sai khi cả *A* và *B* cùng đúng. C không có phép toán này nhưng lại dùng kí hiệu ^ cho phép toán "XOR bít" (chứ không phải là mũ hay XOR logic).

> 1) && 2 có kết quả là 1 (Đúng) do biểu thức luận lý (2 > 1) có kết quả Đúng (1) còn 2 là biểu thức số có giá trị 2 khác 0 nên khi được dùng như giá trị luận lý thì là Đúng nên kết quả là Đúng (tức là 1). Cũng lưu ý, để mô tả điều kiện mà Toán hay viết là 10 > x > 1 thì ta phải viết là `10 > x && x > 1`[17]. Nếu viết `10 > x > 1` thì nếu biến x có giá trị là 2 thì biểu thức `10 > x > 1` có giá trị là 0 (Sai) vì: > có tính kết hợp trái nên `10 > x` được tính trước và có kết quả là 1 (Đúng), sau đó `1 > 1` được tính và có kết quả là 0 (Sai).

Biểu thức kí tự là biểu thức số nguyên mô tả một kí tự bằng mã ASCII của kí tự đó. Chẳng hạn biểu thức `'a' + 1` có giá trị là số nguyên 98 (do hằng kí tự `'a'` có giá trị 97 là mã ASCII của kí tự *a* và 97 + 1 được 98). Nếu được dùng để xác định một kí tự thì `'a' + 1` xác định kí tự *b* do kí tự *b* có mã ASCII là 98 (xem lại Bài 1.7 để nắm rõ).

Không có ranh giới rõ ràng giữa các loại biểu thức gốc số. Chẳng hạn biểu thức `(a > b) * a + (a <= b) * b` sẽ cho giá trị là số lớn nhất trong 2 số *a*, *b*. Thật vậy nếu *a* > *b* thì: biểu thức `a > b` có giá trị là 1 (Đúng) và biểu thức `a <= b` có giá trị là 0 (Sai) nên `(a > b) * a` có giá trị là *a* còn `(a <= b) * b` có giá trị là 0 nên tổng của chúng có giá trị là *a*, chính là số lớn nhất khi *a* > *b*[18]. Ở đây ta đã khéo léo dùng kết hợp (lẫn lộn) biểu thức luận lý với biểu thức số học bằng cách dùng kết hợp các toán tử quan hệ (>, <=) với toán tử số học (*, +). Toàn bộ biểu thức trên được xem là biểu thức số học vì giá trị của nó được dùng như một số nguyên thông thường (là max {*a*, *b*}). Tương tự như vậy, bạn có thể đưa ra biểu thức tính giá trị nhỏ nhất của *a*, *b* được không? tính trị tuyệt đối của *a* được không?

Tương tự, không có ranh giới rõ ràng giữa biểu thức số học và biểu thức kí tự. Nó là gì thì tùy theo mục đích sử dụng như ví dụ sau cho thấy:

```
printf("%d", 'a' + 1);
```

sẽ xuất ra số nguyên 98. Còn:

```
printf("%c", 'a' + 1);
```

sẽ xuất ra kí tự *b*.

MỞ RỘNG 2.1 – Biểu thức địa chỉ

Có một loại biểu thức gốc số nữa mà ta đã gặp và sẽ được tìm hiểu kĩ ở các bài sau đó là *biểu thức địa chỉ*. Đây là biểu thức có giá trị là số nguyên không âm và được dùng như là *địa chỉ* của một *vùng nhớ*. Bạn sẽ thấy rằng mọi dữ liệu phải để trên vùng nhớ để chương trình có thể thao tác,

[17] Không cần đóng ngoặc `(10 > x)` vì toán tử && có độ ưu tiên kém hơn các toán tử so sánh.

[18] Bạn tự phân tích trường hợp *a* = *b* và *a* < *b* để thấy kết quả.

xử lý. Biến là cách thông dụng và đơn giản để ta có thể làm việc với vùng nhớ, đó là vùng nhớ được đặt tên. Tuy nhiên ta có thể không dùng *tên vùng nhớ* mà dùng *vị trí của vùng nhớ* để xác định nó và phương tiện để xác định vị trí của vùng nhớ chính là địa chỉ của vùng nhớ[19]. Mặc dù địa chỉ (và đối tác của nó là *con trỏ*) sẽ được bàn đến ở tầng rất cao sau này nhưng ta thực sự đã gặp chúng:

```
int a;
scanf("%d", &a);
printf("%d\n", a);
```

Đoạn chương trình trên cho nhập một số nguyên vào biến a và xuất số nguyên đó ra. Ở trên ta có &a (trong `scanf`) và a (trong `printf`) đều là biểu thức gốc số. Biểu thức a là biểu thức số học vì ta dùng giá trị như số thông thường. Biểu thức &a là biểu thức địa chỉ với & là *toán tử lấy địa chỉ*, đây là toán tử một ngôi giúp ta lấy về địa chỉ của vùng nhớ của một biến với cú pháp:

&<tên biến>

Giá trị của &a là một số nguyên không âm nhưng ta không dùng nó như số thông thường mà dùng nó để xác định vị trí của vùng nhớ. Sở dĩ ta cần dùng biểu thức địa chỉ trong `scanf` vì `scanf` nhận số nguyên từ người dùng và bỏ số đó vào một vùng nhớ nên ta cần cung cấp địa chỉ (tức là vị trí) của vùng nhớ đó. Ở đây ta bỏ vào vùng nhớ của biến a nên cần cung cấp của địa chỉ của biến a, đó là &a. Ngược lại trong `printf` ta cần xuất ra giá trị đang chứa trong biến a nên ta dùng biểu thức đơn a mà giá trị chính là số đang chứa trong a.

Bạn nghĩ là lệnh sau sẽ xuất ra cái gì?

```
printf("%d\n", &a);
```

còn như lệnh sau đây sẽ thế nào?

```
scanf("%d", a);
```

hay như đoạn mã sau thì sao?

```
int a, b;
printf("%d %d\n", &a, &b);
```

Hãy thử sai, suy ngẫm và tự tìm câu trả lời mà không cần chắc chắn. Ta sẽ thấy chúng thật rõ ràng sau.

[19] Tương tự địa chỉ nhà giúp xác định vị trí của nhà.

BÀI TẬP

Bt 2.1.1 Viết chương trình nhập giá trị thực *x* và tính giá trị của các biểu thức sau bằng cách chỉ dùng các toán tử phù hợp (mà không dùng hàm hay các cấu trúc chọn, lặp):

a) x^{20}.
b) $4x^{64} + 3x^{36} + 2(x - 1)^8 + 1$.
c) $|x|$.
d) sign(*x*) (sign(*x*) kí hiệu cho dấu của *x*: là 1 nếu *x* dương, là 0 nếu *x* là 0 và là -1 nếu *x* âm).

Bt 2.1.2 Vẽ cây biểu thức, tính giá trị bằng cách duyệt cây, tính giá trị "bằng mắt" và chạy thử trong C[20] để đối chiếu cho các biểu thức sau:

a) `3*x*x - 2*x + 1` (với x là biến `int`, đang chứa âm 1 (-1)).
b) `10*x%y + -(100/x - y)` (với x, y là các biến `int`, lần lượt đang chứa 5, 8).
c) `!((x <= 8) && (y > 4))` (với x, y là các biến `int`, lần lượt đang chứa 5, 8).
d) `10 + 3 > 5 && (int) 2.9 || ! (5 / 0)`
e) `1 + 2 || 3 > 4 && 5 < 6 / (int) 7.8 + ! 9 == 10`

Bt 2.1.3 Cho biết 4 lệnh sau đây xuất ra kết quả là gì? Giải thích, so sánh 4 lệnh đó và rút ra qui tắc.

Lệnh 1:
```
if(10 < 11 < 12)
    printf("Dung");
else
    printf("Sai");
```
Lệnh 2:
```
if(12 > 11 > 10)
    printf("Dung");
else
    printf("Sai");
```
Lệnh 3:
```
if(12 > 11 && 11 > 10)
    printf("Dung");
else
    printf("Sai");
```
Lệnh 4:
```
if((12 > 11) * (11 > 10))
    printf("Dung");
else
    printf("Sai");
```

---
[20] Dùng `printf("%d", <biểu thức>)` để xuất giá trị của biểu thức.

BÀI 2.2

Ta bắt đầu bài này bằng toán tử quan trọng nhất của C (cũng như của nhiều ngôn ngữ lập trình khác): *toán tử gán* (=). Toán tử gán là toán tử hai ngôi và *biểu thức gán* có cú pháp:

<l-value> = <r-value>

Nói ngắn gọn, ngữ nghĩa của toán tử gán là: tính giá trị của biểu thức bên phải (<r-value>) và bỏ giá trị đó vào *vùng nhớ* do biểu thức bên trái (<l-value>) chỉ ra[1]. Chi tiết hơn, biểu thức này được lượng giá bằng các bước:

- Tính giá trị của <r-value>.
- Tìm vùng nhớ do <l-value> chỉ đến.
- Giá trị của <r-value> phải phù hợp với kiểu của vùng nhớ mà <l-value> trỏ đến. Nếu không phù hợp thì báo lỗi[2] còn ngược lại thì được phép gán và có thể có chuyển kiểu nếu kiểu không giống nhau[3].
- Giá trị của cả biểu thức gán là <l-value>.

<r-value> và <l-value> đều là các biểu thức. Tuy nhiên một biểu thức muốn là *r-value*[4] thì nó phải *có giá trị*, như vậy đó không thể là *biểu thức void* (*biểu thức không có giá trị*). Một biểu thức để là *l-value* thì còn khó hơn nữa, nó phải chỉ đến một vùng nhớ. Có nhiều cách để chỉ đến vùng nhớ như dùng biểu thức địa chỉ, …[5]. Hiện giờ thì <l-value> là tên biến và vùng nhớ tương ứng là vùng nhớ chứa biến. Nói nôm na <l-value> là biểu thức tương đương với biến. Khi một biểu thức là l-value thì ta có thể lấy giá trị trong vùng nhớ nó chỉ đến làm giá trị của biểu thức nên một l-value cũng là một r-value . Tuy nhiên không phải biểu thức r-value nào cũng có thể dùng như l-value. Hiển nhiên một hằng số thì là r-value nhưng không là l-value.

Ví dụ, nếu a, b là các biến nguyên `int` thì đoạn mã:

```
a = 10.5;
```

---

[1] Chữ *l* và *r* lần lượt là left (trái) và right (phải).

[2] Lỗi ngữ nghĩa.

[3] Chi tiết sẽ rõ sau.

[4] Có chút chơi chữ chỗ này: <r-value> là kí hiệu cho bất kì biểu thức nào có thể dùng như là toán hạng thứ hai (tức là vế phải) của toán tử gán. Biểu thức như vậy gọi là có tính chất r-value. Tuy nhiên ta cũng nói biểu thức nào đó là một <r-value>, hay ngắn gọn là <r-value>, để nói rằng nó có tính chất r-value. Điều đó có nghĩa là viết <r-value> cũng như viết r-value. (… Hiểu được chết liền!:))

[5] Mà ta sẽ biết sau.


```
b = a;
b = b * 2;
```

sẽ bỏ giá trị 10 (10.5 là giá trị thực được chuyển thành giá trị nguyên 10) vào vùng nhớ của biến a. Như vậy a chứa giá trị 10. Sau đó giá trị của biến a (10) được bỏ vào vùng nhớ của biến b. Như vậy b cũng chứa giá trị 10. Sau đó giá trị của biểu thức `b * 2` là 20 (vì b đang chứa giá trị 10 nên giá trị của b là 10) được bỏ vào biến b. Ta nói rằng b đã thay đổi (hay *cập nhật*) giá trị mới là 20 thay cho giá trị cũ là 10. Ở lệnh thứ nhất thì 10.5 là r-value và a là l-value. Lệnh thứ 2 đặc biệt hơn, ta thấy vế trái và phải của toán tử gán đều là biến nhưng ý nghĩa hoàn toàn khác nhau: a là l-value nhưng ở bên phải thì được dùng như là r-value có giá trị là 10 (là giá trị đang chứa trong biến a) còn b bên trái là l-value chỉ đến vùng nhớ của biến b. Ở lệnh thứ 3 cũng vậy, thậm chí cùng biến b, nhưng b bên trái là l-value có giá trị 10 còn b bên phải là r-value có giá trị là giá trị đang chứa trong biến b. Như vậy cùng là biến nhưng khi để bên phải toán tử gán thì có ngữ nghĩa là giá trị chứa trong biến đó (và do đó là r-value) còn khi để bên trái thì có ngữ nghĩa là vùng nhớ của biến (và do đó là l-value).

Thật ra 2 lệnh đầu có thể được viết gọn thành:

```
b = a = 10.5;
```

Nên nhớ rằng toán tử gán (=) cũng là toán tử, biểu thức gán cũng là biểu thức. Hơn nữa *toán tử gán có tính kết hợp phải* nên biểu thức: `b = a = 10.5` là biểu thức phức, có thể được viết rõ hơn là `b = (a = 10.5)` trong đó biểu thức `a = 10.5` được tính trước, dẫn đến a chứa giá trị 10, hơn nữa, biểu thức này có giá trị là a. Sau đó `b = a` được tính, ở đây thì a (là l-value) được dùng như r-value có giá trị là 10 (giá trị đang chứa trong a) nên dẫn đến b cũng chứa giá trị 10. Thật ra, cả 3 lệnh trên có thể gom vào một lệnh duy nhất:

```
b = (b = a = 10.5) * 2;
```

Cũng cần lưu ý là phải có cặp ngoặc như trên vì toán tử = có độ ưu tiên thấp hơn toán tử *[6]. Cũng lưu ý, lệnh sau là hợp lệ:

```
(b = a) = 10.5;
```

nhưng `(b = a)` sẽ được thực hiện trước, khi đó b sẽ chứa giá trị đang chứa trong a, kết quả của biểu thức này là b, sau đó `b = 10.5` được thực hiện và kết quả là b chứa 10 (10.5 được chuyển thành 10). Như vậy chỉ có b chứa giá trị mới còn a thì vẫn giữ giá trị cũ.

Như đã thấy, biến có thể được dùng như là như là r-value hay l-value. Mọi biểu thức l-value đều có thể dùng như là r-value (khi đã có vùng nhớ thì

---

[6] Không khuyến khích cách viết này vì khó đọc và có thể là nguồn lỗi khó phát hiện.

r-value chính là giá trị trong vùng nhớ đó). Tuy nhiên rất ít dạng biểu thức có thể dùng như cả hai (như trường hợp của biến). Hầu hết các biểu thức chỉ dùng được như là r-value. Chẳng hạn các lệnh sau đều bị lỗi[7]:

```
10 = a;
b * 2 = b;
```

`10` hay `b * 2` đều không chỉ đến vùng nhớ nên chúng không là các l-value, chúng chỉ là các r-value với giá trị là 10 hay là 2 lần giá trị đang chứa trong `b`.

C có rất nhiều toán tử, nhiều trong đó mang lại sự tiện lợi (mặc dù không thực sự cần thiết). Chẳng hạn thay vì viết `b = b + 2` ta có thể viết gọn hơn là `b += 2` Bên cạnh += ta có các toán tử gán khác với ý nghĩa tương tự là: `-=`, `*=`, `/=`, `%=`, … Những toán tử này được gọi là *toán tử gán kép* để phân biệt với toán tử = gọi là *toán tử gán đơn*. Như vậy thay vì viết `a = a * (<biểu thức>)` ta có thể viết thành `a *= <biểu thức>`. Cũng lưu ý là biểu thức `b += 2` có giá trị là `b` (là một l-value) nên ta có biểu thức hợp lệ là `(b += 2) += 2` nhưng biểu thức `b += (2 += 2)` thì không hợp lệ vì 2 không là l-value. Cũng như toán tử gán đơn, *các toán tử gán kép cũng có tính kết hợp trái* nên biểu thức `i += j += k` được hiểu là `i += (j += k)`, biểu thức `i += 2 += k` không hợp lệ, còn biểu thức `(i += 2) += k` thì hợp lệ.

Một điều cực kì quan trọng nữa về các toán tử gán là chúng thuộc nhóm các toán tử có *hiệu ứng lề*. Một toán tử được gọi là có hiệu ứng lề nếu việc lượng giá biểu thức có chứa toán tử đó có khả năng làm thay đổi *trạng thái chương trình*, mà cụ thể là thay đổi giá trị của các biến trong chương trình[8]. Chẳng hạn nếu `a`, `b` là các biến nguyên đang chứa giá trị tương ứng 10, 20 thì việc lượng giá biểu thức `a + b` (được giá trị là 30) không làm thay đổi giá trị của biến `a`, `b` (và các biến khác) nhưng việc lượng giá biểu thức `a = b` (được giá trị là 20) làm thay đổi giá trị của biến `b` (`b` chứa giá trị mới là 20). Do đó toán tử + là không có hiệu ứng lề còn toán tử = là có hiệu ứng lề. *Các toán tử số học, quan hệ, logic là các toán tử không có hiệu ứng lề*. *Các toán tử gán (đơn và kép) là các toán tử có hiệu ứng lề*.

Hai toán tử có hiệu ứng lề nữa là *toán tử tăng* (++) và *toán tử giảm* (`--`). Mục đích của hai toán tử này là để viết gọn. Thay vì viết `a = a + 1` bạn có thể viết `a += 1` hay ngắn hơn nữa là `a++`. Tương tự, thay vì viết `a = a - 1` bạn có thể viết `a -= 1` hay ngắn hơn nữa là `a--`. Hai toán tử này đều là một ngôi và hiển nhiên là toán hạng của nó phải là l-value (`10++` sẽ báo lỗi vì 10 không là l-value). Khác với các toán tử một ngôi khác thường được viết ở dạng *tiền tố* (toán tử viết trước toán hạng) hoặc dạng *hậu tố* (toán tử viết

[7] Lỗi cú pháp hoặc lỗi ngữ nghĩa (tùy compiler).

[8] Trạng thái chương trình còn bao gồm trạng thái nhập, xuất, … mà ta sẽ tìm hiểu sau.

sau toán hạng), chúng có thể được viết theo hai cách với hai cách lượng giá khác nhau:

- *Tiền tố*: ++a, a chứa giá trị mới là giá trị cũ cộng thêm 1 (hiệu ứng lề) và giá trị của cả biểu thức ++a là a (là l-value và nếu dùng như r-value thì là giá trị mới chứa trong a).
- *Hậu tố*: a++, a chứa giá trị mới là giá trị cũ cộng thêm 1 (hiệu ứng lề) và giá trị của cả biểu thức a++ là giá trị cũ của a (r-value).

Như vậy nếu a đang chứa giá trị 1 thì ++a có giá trị là 2 còn a++ có giá trị là 1 và a đều chứa 2 sau khi lượng giá biểu thức. Tương tự `--a` có giá trị là 0 còn `a--` có giá trị là 1 và a đều chứa 0 sau khi lượng giá biểu thức. Như vậy, `a = a + 1` hay `a += 1` có thể được thay bằng ++a (chứ không phải là a++) nếu ta có nhận về giá trị của biểu thức. Lưu ý là mặc dù thường dùng với biến nguyên nhưng hai toán tử ++, `--` cũng có thể dùng với biến thực.

Bạn có nhớ nổi những qui tắc này không?[9] Lời khuyên: *bạn không nên dùng 2 toán tử này khi nhận lại giá trị của nó*. Chẳng hạn bạn có thể viết lệnh: `i++;` thay cho `i = i + 1;` như trong cấu trúc lặp `for` vì bạn không nhận về giá trị (chỉ dùng hiệu ứng lề để tăng giá trị của `i`). Còn lại ta nên tránh dùng khi nhận lại giá trị. Chẳng hạn nếu `i`, `j`, k đang chứa lần lượt các giá trị là 1, 2, 3. Bạn có biết giá trị của `i`, `j`, k sau khi thực hiện lệnh: `k = ++i + j++;` là bao nhiêu không?[10] Cũng lưu ý các *toán tử tăng, giảm có độ ưu tiên cao nhất*: chẳng hạn bạn có thể viết `-i--` nhưng nếu viết `(-i)--` thì báo lỗi đấy nhé. Tại sao vậy?

Tới đây thì bạn đã biết một số (kha khá) các toán tử của C. Tuy nhiên còn nhiều toán tử khác nữa mà bạn sẽ học. Phụ lục A.6 cung cấp danh sách các toán tử C cùng với độ ưu tiên và tính kết hợp của chúng.

Một điều cực kì quan trọng với biểu thức mà bạn cần nhớ: *mọi biểu thức trong C đều có thể chuyển thành lệnh bằng cách thêm dấu* `;` *vào cuối*. Lệnh này do đó gọi là *lệnh biểu thức*. Nó còn được gọi là *lệnh đơn* và là dạng lệnh đơn giản nhất[11]. Như vậy lệnh đơn có cú pháp:

<biểu thức>;

Mọi biểu thức đều có thể chuyển thành lệnh biểu thức nhưng chỉ có các biểu thức có hiệu ứng lề mới có ý nghĩa vì giá trị của biểu thức sẽ bị bỏ đi. Chẳng hạn: `a + 1;` là lệnh đơn hợp lệ nhưng nó không có ý nghĩa (ta có thể bỏ nó đi mà không ảnh hưởng đến chương trình)[12] do toán tử + không có hiệu ứng lề và giá trị của `a + 1` cũng không được dùng đến. Tuy nhiên lệnh: `a++;` thì có ý nghĩa do toán tử ++ có hiệu ứng lề (giá trị của a sẽ được tăng

---

[9] Tôi không nhớ nổi.
[10] Lần lượt là 2, 3, 4.
[11] Thực ra lệnh đơn giản nhất là lệnh rỗng, nó chỉ là `;`
[12] Còn gọi là lệnh không làm gì, đúng hơn là làm chuyện không công:)

thêm 1). Hơn nữa ta thường gọi các lệnh biểu thức của toán tử gán là *lệnh gán* như lệnh `a = a + 1;` hay lệnh `a += 1;`.

MỞ RỘNG 2.2 – Lượng giá biểu thức có hiệu ứng lề

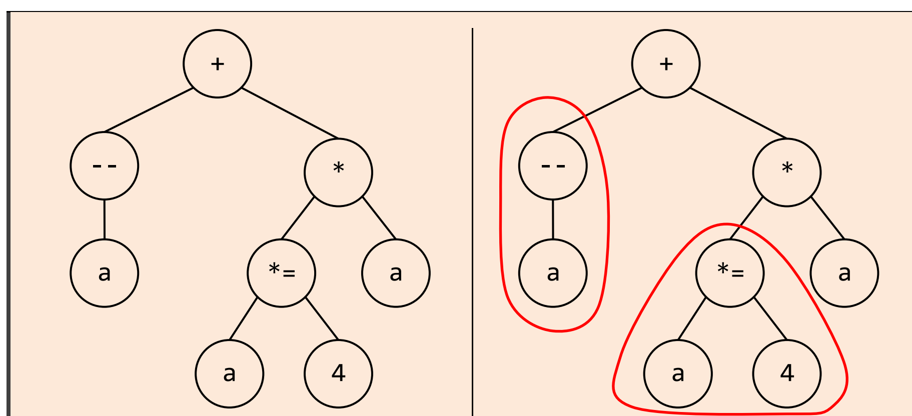
Các toán tử có hiệu ứng lề gây nhiều khó khăn khi lượng giá biểu thức. Khi lượng giá biểu thức phức thì chỉ có duy nhất một qui tắc là *các biểu thức con phải được lượng giá trước biểu thức cha*. Chẳng hạn: *A* + *B* * *C* thì *B*, *C* phải được tính trước khi tính *B* * *C* (gọi là biểu thức *D*); *A* và *D* phải được tính trước khi tính *A* + *D* (gọi là *E*). Như vậy ta có thể tính theo thứ tự các biểu thức: *B*, *C*, *D*, *A*, *E* hay *A*, *B*, *C*, *D*, *E* hay *C*, *B*, *D*, *A*, *E*; … đều thỏa mãn *B*, *C* trước *D* và *A*, *D* trước *E*. Nếu các biểu thức đều không có hiệu ứng lề thì việc thực hiện theo cách nào không quan trọng vì đều cho ra cùng một kết quả, nhưng nếu có hiệu ứng lề thì thứ tự thực hiện các biểu thức sẽ có thể cho giá trị khác nhau. Chẳng hạn với biểu thức `(a + 1)*(a + 2)` thì hai phép cộng phải được thực hiện trước phép nhân nhưng ta có thể chọn làm phép cộng thứ nhất trước, phép cộng thứ hai sau hoặc ngược lại[13]. Giả sử a là biến `int` đang chứa giá trị 0 thì biểu thức trên có thể được lượng giá bằng hai cách: tính `a + 1` trước được 1, sau đó tính `a + 2` được 2 và nhân 2 kết quả lại được 1*2 là 2 hoặc tính `a + 2` trước được 2, sau đó tính `a + 1` được 1 và nhân 2 kết quả lại được 1*2 cũng là 2. Cả hai cách lượng giá đều cho kết quả như nhau vì biểu thức không có hiệu ứng lề. Ngược lại biểu thức có hiệu ứng lề như `(++a)*(a = 2)` thì hai cách lượng giá sẽ cho hai kết quả khác nhau. Thật vậy, cũng giả sử a là biến `int` đang chứa giá trị 0 thì theo cách thứ nhất: tính ++a trước được a và a chứa giá trị mới là 1, sau đó tính `a = 2` được a và a chứa giá trị mới là 2, sau đó nhân hai kết quả lại được a*a là 2 * 2 là 4 do a đang chứa 2. Còn nếu lượng giá theo cách thứ hai: tính `a = 2` trước được a và a chứa giá trị mới là 2, sau đó tính ++a được a và a chứa giá trị mới là 3, sau đó nhân hai kết quả lại được a*a là 3 * 3 là 9 do a đang chứa 3.

Tương tự, xét biểu thức `--a + (a *= 4)*a`. Biểu thức này có những cách lượng giá nào khác nhau? Nếu khó thấy bạn có thể vẽ cây biểu thức của biểu thức này trước như Hình (a) sau. Lưu ý là *cây biểu thức là duy nhất cho biểu thức*.

Hình 2.2.1 – Cây biểu thức của biểu thức `--a + (a *= 4)*a`

Hình (a)	Hình (b)

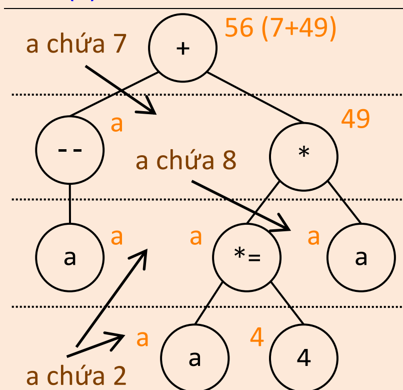
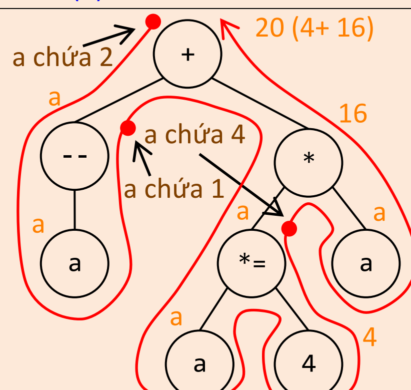
[13] Điều này không liên quan gì đến tính kết hợp trái của phép cộng. Trong biểu thức `a + 1 + 2` thì phép cộng thứ nhất phải được làm trước phép cộng thứ hai do tính kết hợp trái của toán tử +, ta không có lựa chọn khác.



Từ cây biểu thức và với qui tắc là nút con phải được tính trước nút cha thì ta thấy có hai cách lượng giá khác nhau liên quan đến toán tử có hiệu ứng lề (là `--` và `*=`) mà cụ thể là tính `--a` trước hay `a *= 4` trước (hai biểu thức khoanh tròn ở Hình (b) trên). Nếu a đang chứa giá trị 2 thì giá trị của biểu thức trên là bao nhiêu? Theo cách lượng giá trong bài trước (qui tắc tính từ dưới cùng lên, từ trái qua phải) thì ta có giá trị của biểu thức là 56 (như Hình (a) dưới), tuy nhiên C lại tính ra giá trị của biểu thức đó là 20. Chẳng hạn bạn có thể chạy đoạn mã sau để thấy:

```
int a = 2;
printf("%d\n", --a + (a *= 4)*a);
```

Hình 2.2.2 – Duyệt cây biểu thức của biểu thức `--a + (a *= 4)*a`

Hình (a)	Hình (b)
	

Như vậy cách lượng giá đơn giản bằng cách duyệt cây biểu thức từ dưới lên, trái qua phải như bài trước không thực sự được compiler dùng và nó có thể không còn đúng khi biểu thức có hiệu ứng lề. Vậy cách duyệt đúng là gì? Khá đơn giản, khi có nhiều cách lượng giá phù hợp với cây biểu thức (tức là phù hợp với độ ưu tiên và tính kết hợp cũng như các cặp

ngoặc) thì đa số compiler sẽ lựa chọn *tính biểu thức con bên trái trước*. Trước hết, ta thử làm "bằng mắt" như bài trước. Trong biểu thức `--a + (a *= 4)*a` thì có thể làm `--a` trước hoặc `a *= 4` trước đều được[14]. Ở đây ta chọn làm `--a` trước vì nó nằm bên trái `a *= 4` do đó biểu thức `--a + (a *= 4)*a` sẽ thành `a + (a *= 4)*a` và a chứa giá trị 1. Sau đó `a + (a *= 4)*a` sẽ thành `a + a*a` và a chứa 4. Sau đó `a + a*a` sẽ thành `a + 16` và `a + 16` sẽ thành 20 có giá trị là 20 (và a chứa 4). Điều này tương ứng với cách duyệt cây ở Hình (b) trên[15]. Xuất phát từ nút gốc, ta đi theo đường mũi tên ngoằn ngèo, đi qua mỗi nút 2 lần (một lần ở bên trái, một lần ở bên phải nút)[16] và ta sẽ lượng giá nút ở lần thứ hai (lần ở bên phải nút). Khi lượng giá nút ứng với toán tử có hiệu ứng lề thì trạng thái chương trình có thể thay đổi (các biến có thể nhận giá trị mới). Ví dụ ở trên, với biến a đang chứa 2, xuất phát từ nút gốc (nút +), ta qua nút `--` lần thứ nhất, rồi qua nút a lần thứ nhất, rồi qua nút a lần thứ hai, khi đó ta lượng giá nút a (được l-value a), rồi qua nút `--` lần thứ hai, khi đó ta lượng giá nút này (được l-value a với hiệu ứng lề là a chứa giá trị mới là 1) với trạng thái chương trình thay đổi, …, qua nút *= lần thứ hai, lượng giá nút này (được l-value a với hiệu ứng lề là a chứa giá trị mới là 4), …, qua nút * lần thứ hai, lượng giá nút này (được `a*a` và a đang chứa 4 nên được 16), qua nút + lần thứ hai, lượng giá nút này (được `a + 16` và a đang chứa 4 nên được 20), kết thúc với giá trị của nút gốc là 20, chính là giá trị của biểu thức[17].

BÀI TẬP

Bt 2.2.1 Giả sử a là biến `int`, x là biến `double`. Cho biết giá trị của a, x sau khi thực hiện lệnh đơn sau:

```
x = a = 1.5;
```

Bt 2.2.2 Giả sử a, b, c là các biến `int` lần lượt chứa các giá trị 3, 6, 9. Cho biết giá trị của các biến sau khi thực hiện lệnh đơn sau:

```
a = c = (b = a + 2) - (a = 1);
```

Bt 2.2.3 Giả sử a, b là các biến `int`. Các cặp lệnh sau có tương đương nhau không? Giải thích?

---

[14] Cặp ngoặc tròn bọc `a *= 4` chỉ nói rằng *= phải được tính trước rồi mới tới * và sau đó mới tới +.

[15] Cách duyệt này được gọi là *duyệt sau* (post-order). (Xem phần Post-order trong bài viết "Tree traversal" trên Wiki, https://en.wikipedia.org/wiki/Tree_traversal)

[16] Đúng ra ta sẽ đi qua mỗi nút 4 lần: trên, trái, dưới, phải của nút. Nhưng ở đây ta chỉ tính 2 lần: trái và phải thôi.

[17] Tôi thật tình không mong đợi là bạn sẽ hiểu hết phần này. Nói chung, bạn chỉ có thể hiểu rõ các phần mở rộng khi bạn luyện xong tầng phù hợp sau đó.

a) `a *= b + 1;` với `a = a*b + 1;`
b) `(a += 2) *= b;` với `(a += 2) = (a += 2)*b;`

Bt 2.2.4 Giả sử a, b, c, d, e, f là các biến `int` lần lượt chứa các giá trị 1, 2, 3, 4, 5, 6. Cho biết các biểu thức sau có lỗi không? Nếu không thì cho biết kết quả của biểu thức và giá trị của các biến sau khi lượng giá biểu thức:

```
a) a += b = c++ * ++d - e *= --f
b) a += b = c++ * ++d - (e *= --f)
c) a += (b = c)++ * ++d - (e *= --f)
d) (a += b = c)++ * ++d - (e *= --f)
e) (a += b = c)++ * (++d - (e *= --f))
f) (a += b = c)++ * ++(d - (e *= --f))
```

Bt 2.2.5 Cho biết kết quả chạy 4 đoạn mã sau? Giải thích, so sánh 4 đoạn mã đó và rút ra qui tắc.

Mã 1:

```
int x = 1;
if(x = 0)
    printf("Dung");
else
    printf("Sai");
```

Mã 2:

```
int x = 1;
if(x = 2)
    printf("Dung");
else
    printf("Sai");
```

Mã 3:

```
int x = 1;
if(x == 2)
    printf("Dung");
else
    printf("Sai");
```

Mã 4:

```
int x = 1;
if(0 = x)
    printf("Dung");
else
    printf("Sai");
```

BÀI 2.3

Hãy xem lại cách tìm nghiệm của đa thức bậc hai trong Bài 1.5.

- Cho đa thức bậc hai: $ax^2 + bx + c$ ($a \neq 0$).
- Tính $\Delta = b^2 - 4ac$.
- Nếu $\Delta < 0$ thì đa thức vô nghiệm.
- Nếu $\Delta = 0$ thì đa thức có một nghiệm $x = \frac{-b}{2a}$.
- Nếu $\Delta > 0$ thì đa thức có hai nghiệm $x_1 = \frac{-b+\sqrt{\Delta}}{2a}, x_2 = \frac{-b-\sqrt{\Delta}}{2a}$.

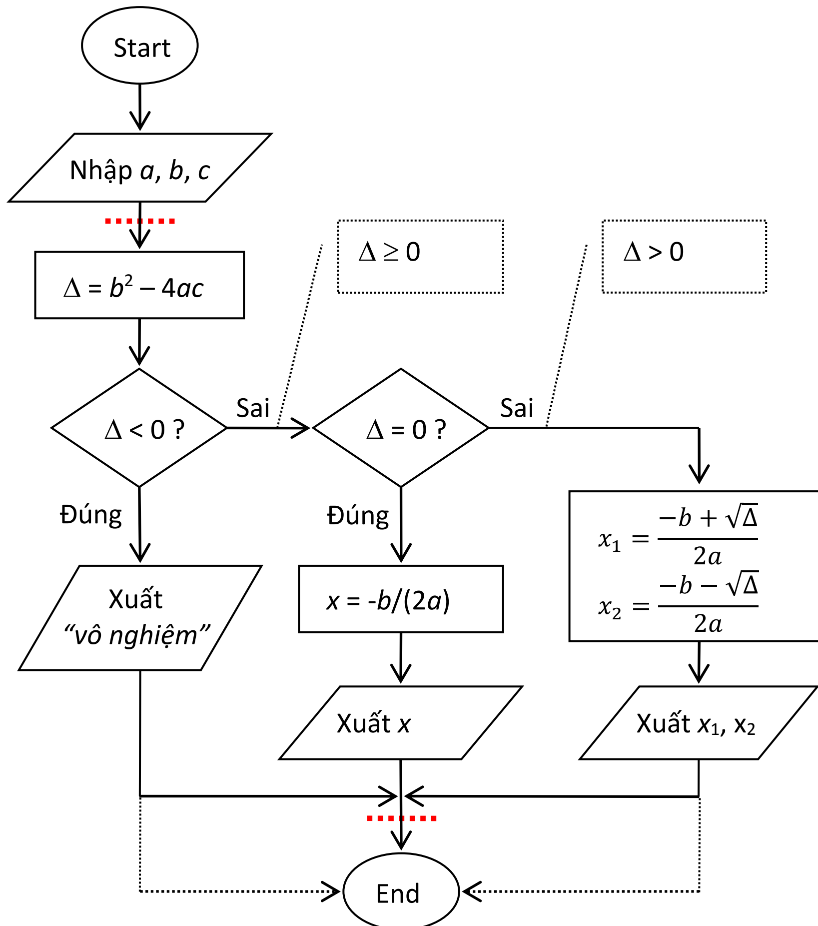
Ở trên là mô tả bằng ngôn ngữ tự nhiên (tiếng Việt) pha với ngôn ngữ Toán (các kí hiệu) cách tìm nghiệm của đa thức bậc hai. Ta cũng thường gọi ngôn ngữ đó là *mã giả*. Ta thậm chí đã viết chương trình C bằng cách mô tả nó trong ngôn ngữ C (Mã 1.5.1). Ở đây tôi sẽ dùng một ngôn ngữ nữa để mô tả cách làm trên, đó là *lưu đồ*.

Lưu đồ trong Hình 2.3.1 mô tả các bước thực hiện một cách rõ ràng: gồm các công việc nào và thứ tự các công việc ra sao (công việc này xong thì tới công việc nào). Các công việc được mô tả trong các hình ellipse, hình bình hành, hình thoi, và hình chữ nhật mà ta gọi là các *nút*. *Mũi tên* từ nút *A* đến nút *B* cho thấy sau khi thực hiện xong công việc của nút *A* thì chuyển qua thực hiện công việc của nút *B*. Khi đó *B* còn được gọi là *nút kế tiếp* của *A*. Mũi tên còn được gọi là *cạnh*. Khi đó nó còn được gọi là *cạnh ra* của nút *A* và *cạnh vào* của nút *B*. Các nút ellipse gọi là *nút đầu cuối*. Quá trình thực thi bắt đầu từ nút đầu cuối có chữ Start (hay Bắt đầu) và kết thúc khi thực thi đến nút đầu cuối có chữ End (hay Kết thúc). Các nút hình bình hành gọi là *nút nhập xuất*, nó mô tả việc lấy dữ liệu từ người dùng (hay từ nguồn khác) với nút có chữ Nhập và mô tả việc xuất dữ liệu ra màn hình (hay ra nguồn khác) với nút có chữ Xuất. Nút hình thoi gọi là *nút lựa chọn*, bên trong nút là một biểu thức luận lý, và có hai cạnh đi ra với chữ Đúng (hay True/Yes) và Sai (hay False/No). Khi thực thi đến nút này, nếu biểu thức luận lý có giá trị là Đúng thì sẽ chuyển đến thực thi nút do cạnh Đúng chỉ tới, còn nếu là Sai thì sẽ chuyển đến thực thi nút do cạnh Sai chỉ tới. Nút hình chữ nhật gọi là *nút thao tác*, mô tả các loại công việc còn lại (không phải là bắt đầu/kết thúc, nhập/xuất hay lựa chọn) mà thường là tính toán hay thao tác nào đó. Ngoài ra ta còn có thể ghi chú lên sơ đồ bằng cách dùng *nút ghi chú* mà mục đích chỉ để ghi chú thêm vào lưu đồ chứ không phải là công việc được thực hiện.

Yêu cầu khi *thiết kết lưu đồ* (tức là vẽ lưu đồ) cho một công việc nào đó là phải rõ ràng và xác định. Các nút có thể dùng kí hiệu toán hay thậm chí

các chữ của ngôn ngữ tự nhiên nhưng phải ngắn gọn, rõ ràng. Xác định nghĩa là khi thực thi xong một nút thì sẽ chuyển sang thực thi đúng một nút kế tiếp. Điều đó có nghĩa là mỗi nút có đúng một cạnh ra, trừ nút quyết định có đúng hai cạnh ra (một với chữ Đúng và một với chữ Sai). Riêng nút kết thúc thì không có cạnh ra vì việc thực thi kết thúc khi đến nút Kết thúc. Cũng lưu ý là một nút có thể có nhiều cạnh vào và trong trường hợp đó ta hay vẽ các cạnh tụ lại một điểm rồi cùng đi vào nút như cách vẽ các cạnh vào của nút End ở Hình 2.3.1 (mà lẽ ra phải vẽ như các đường đứt nét trên hình).

Hình 2.3.1 – Lưu đồ tìm nghiệm đa thức bậc hai $ax^2 + bx + c$ ($a \neq 0$)



Cũng lưu ý là ta có thể dùng các biến (với ý là chứa kết quả tính toán) mà không cần để ý đến việc khai báo biến như trong C. Ở trên thì các biến nhập (a, b, c), biến xuất (x, x_1, x_2) và biến trung gian (Δ) đã được dùng mà không khai báo. Ta cũng hiểu ngầm là chúng chứa các giá trị thực.

Quá trình thực thi bắt đầu từ nút Start, đi theo các cạnh qua các nút và kết thúc khi tới nút End. Nếu lưu đồ có nút lựa chọn thì có thể có nhiều cách đi từ nút Start đến nút End mà mỗi cách đi (còn gọi là *đường đi*) là mỗi hướng thực hiện khác nhau. Chẳng hạn với lưu đồ trên ta thấy có 3 hướng thực hiện ứng với 3 cách đi từ nút Start đến nút End tùy theo giá trị của Δ. Mỗi hướng đi còn gọi là một *luồng thực thi* nên sơ đồ này được gọi là sơ đồ luồng thực thi.

Lưu đồ mô tả tất cả các hướng thực hiện nên có thể xem là *kịch bản thực hiện*. Việc chạy hay *thực thi lưu đồ* là việc thực hiện lưu đồ một cách thực tế. Lưu đồ có thể có nhiều hướng nhưng khi thực thi nó sẽ theo một hướng cụ thể nào đó thôi (tùy theo giá trị tại các nút luận lý mà điều này thường tùy thuộc vào giá trị nhập cho các biến). Chẳng hạn ta thử chạy lưu đồ trên:

- Bắt đầu từ nút Start, theo cạnh ra của nó ta đến nút Nhập. Giả sử người dùng nhập *a* = 1, *b* = 2, *c* = 2. Theo cạnh ra của nó ta đến nút thao tác, tính Δ được -4. Theo cạnh ra của nó đến nút lựa chọn, do $\Delta < 0$ là đúng (-4 < 0) nên đi theo cạnh Đúng ta đến nút Xuất. Thực thi nút này xong (xuất ra thông báo "vô nghiệm"), ta đến nút End và kết thúc.

Một trường hợp chạy khác:

- Bắt đầu từ nút Start, theo cạnh ra của nó ta đến nút Nhập. Giả sử người dùng nhập *a* = 1, *b* = 2, *c* = 1. Theo cạnh ra của nó ta đến nút thao tác, tính Δ được 0. Theo cạnh ra của nó đến nút lựa chọn, do $\Delta < 0$ là sai (0 = 0) nên đi theo cạnh Sai ta đến nút lựa chọn kế tiếp, do $\Delta = 0$ là đúng (0 = 0) nên đi theo cạnh Đúng ta đến nút thao tác, tính $x = -b/(2a)$ được x là -1. Theo cạnh ra của nó đến nút Xuất. Thực thi nút này xong (xuất ra nghiệm $x = -1$), ta đến nút End và kết thúc.

Hay một trường hợp chạy khác:

- Bắt đầu từ nút Start, theo cạnh ra của nó ta đến nút Nhập. Giả sử người dùng nhập *a* = 1, *b* = 2, *c* = 0. … (Đuối, bạn chạy tiếp nhé:))

Hiển nhiên là có vô số khả năng thực thi khác nhau do có vô số cách nhập giá trị cho *a*, *b*, *c*. Tuy nhiên, qua phân tích lưu đồ, ta thấy chỉ có 3 cách (3 hướng hay 3 luồng) thực thi khác nhau mà thôi. Như vậy, lưu đồ là một công cụ (hay ngôn ngữ) mô tả công việc rất tốt, vừa trực quan, sinh động, dễ hiểu (hơn so với các ngôn ngữ lập trình như C) lại vừa rõ ràng chính xác (hơn ngôn ngữ tự nhiên), lại có thể giúp phân tích được nhiều vấn đề khác nhau (như các hướng thực thi) của công việc. Tuy nhiên nó cũng có nhược điểm là rất tốn công để ngồi vẽ lưu đồ so với việc mô tả bằng C[1]. Điều quan trọng

---

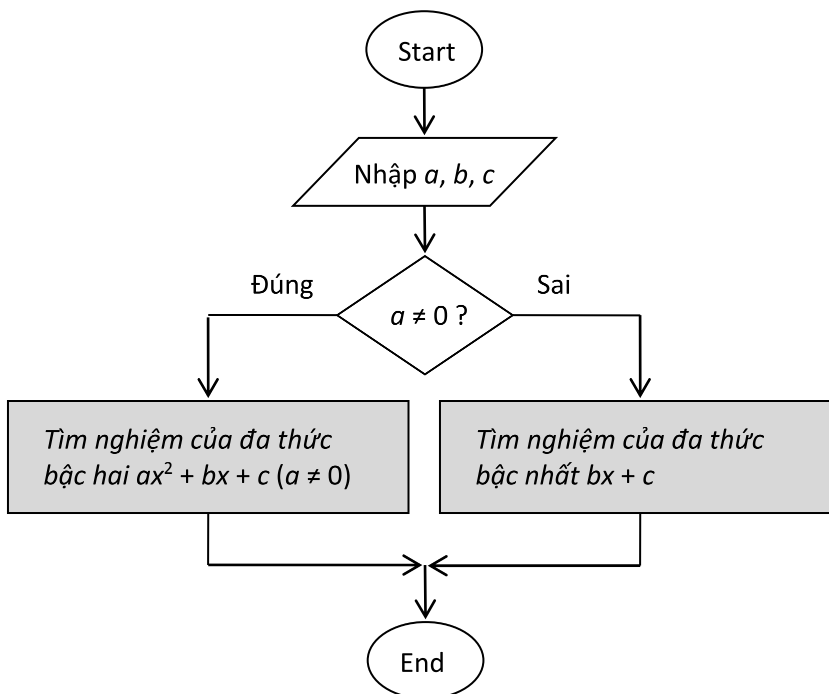
[1] Thực tế là tôi đã tốn không dưới một giờ để vẽ cái lưu đồ trên trong Word nhưng chỉ cần vài phút là viết xong chương trình bằng mã C.

hơn là máy vẫn không hiểu được công việc khi mô tả bằng lưu đồ[2] (cũng như ngôn ngữ tự nhiên) cho nên ta cũng phải lập trình bằng ngôn ngữ lập trình như C. Tuy nhiên rất đáng giá khi dùng lưu đồ trong quá trình viết chương trình nhất là với những người mới học lập trình. Các bước viết chương trình nên là: *xác định bài toán, nghĩ về thuật toán, mô tả rõ thuật toán với lưu đồ, cài đặt bằng ngôn ngữ lập trình*. Tức là: *nghĩ trong đầu, viết ra giấy, làm trên máy*.

Với bài toán tìm nghiệm của đa thức bậc hai ta đã xác định bài toán, đã suy nghĩ cách giải (trong Bài 1.5), cũng đã vẽ lưu đồ (Hình 2.3.1). Vậy ta viết mã C từ lưu đồ trên như thế nào? Mặc dù ta đã "làm đại" trong Bài 1.5 (Mã 1.5.1) nhưng một cách hệ thống ta sẽ học cách chuyển lưu đồ trên sang mã C trong bài sau.

Ở trên ta đã yêu cầu hệ số a khác không. Trong trường hợp tổng quát để tìm nghiệm của đa thức $ax^2 + bx + c$ thì trường hợp $a = 0$ sẽ đưa về đa thức bậc nhất $bx + c$. Như vậy ta có lưu đồ để tìm nghiệm của đa thức $ax^2 + bx + c$ nói chung như Hình 2.3.2.

Hình 2.3.2 – Lưu đồ tìm nghiệm đa thức $ax^2 + bx + c$



[2] Thực ra thì đã có những công cụ giúp làm việc này, tức là lập trình bằng lưu đồ nhưng còn nhiều hạn chế, nhất là với các chương trình lớn, phức tạp.

Trong lưu đồ trên ta đã dùng một loại nút đặc biệt: *hộp đen*[3]. Nút hộp đen là nút mô tả một công việc mà ta chưa làm rõ. Chẳng hạn nút hộp đen ứng với công việc "Tìm nghiệm của đa thức bậc nhất *bx* + *c*" thì ta chưa làm rõ là sẽ làm như thế nào. Tuy nhiên ta đã biết công việc chung của nút đó là gì (tìm nghiệm của đa thức bậc nhất) với đầu vào đã có là *b*, *c* (nghĩa là không nhập *b*, *c* nữa). Nút hộp đen thì không thực thi được (do chưa biết thực thi thế nào). Ta cần làm rõ công việc của nút hộp đen nghĩa là thiết kế lưu đồ cho công việc của nút hộp đen. Việc này được gọi là *phân giải* nút hộp đen. Lưu đồ cho nút hộp đen con gọi là *lưu đồ con* (so với *lưu đồ cha* là lưu đồ của toàn bộ công việc bên ngoài). Khi đã thiết kế lưu đồ của nút hộp đen, ta có thể "*ghép*" nó vào lưu đồ lớn hoặc cũng có thể để riêng nó mà khi chạy lưu đồ tới nút hộp đen thì ta chạy vào lưu đồ con của nút hộp đen đó[4].

Lợi ích của lưu đồ với các hộp đen là ta thấy được bố cục toàn thể của công việc (còn gọi là khung công việc và nếu được cài đặt xuống thành chương trình thì ta gọi là *khung chương trình*). Tuy nhiên lưu đồ với các hộp đen là chưa hoàn chỉnh và không chạy được. Ta cần phân giải tất cả các nút hộp đen để được lưu đồ hoàn chỉnh và chạy được.

Nghiệm của đa thức bậc nhất *bx* + *c* được tính như sau[5]:

- Nếu $b \neq 0$: có một nghiệm là $x = -c/b$.
- Nếu $b = 0$:
    - Nếu $c = 0$: vô số nghiệm.
    - Nếu $c \neq 0$: vô nghiệm.

Từ cách tính trên bạn hãy vẽ lưu đồ cho việc tìm nghiệm của đa thức bậc nhất *bx* + *c* với lưu ý là ta không có nút nhập cho *b*, *c* vì *b*, *c* đã có rồi (đã được nhập trước đó ở lưu đồ lớn). Đối với nút hộp đen "tìm nghiệm của đa thức bậc hai" thì ta đã có lưu đồ cho nó trước đó (Hình 2.3.1). Bạn hãy ghép vào lưu đồ lớn nhé. Cũng nhớ là không có nút nhập *a*, *b*, *c* vì ta đã nhập trước đó. Hơn nữa cần cắt bỏ các nút Start và End của lưu đồ con[6]. Nghĩa là bạn cắt bỏ phần trên của lưu đồ con tại đường gạch gạch ở trên và ghép với cạnh vào của nút hộp đen và cắt bỏ phần dưới của lưu đồ con tại đường gạch gạch ở dưới và ghép với cạnh ra của nút hộp đen[7].

Trong Bài 1.5 ta cũng đã viết chương trình C cho người dùng nhập các số nguyên và tính tổng cho đến khi người dùng nhập số 0 thì dừng và xuất ra tổng. Ở đây tôi sẽ thiết kế lưu đồ cho công việc đó như hình sau.

---

[3] Tôi chỉ tô màu xám vì nếu tô đúng nghĩa đen, màu đen, thì không thấy chữ bên trong:)

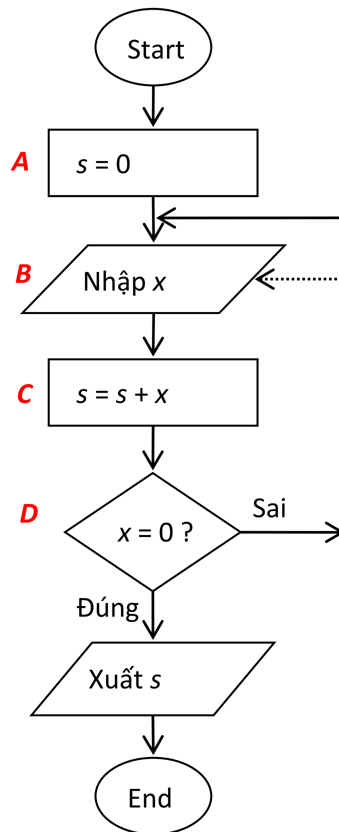
[4] Cách làm này tương ứng với việc tạo và chạy các chương trình con mà ta sẽ được học sau.

[5] Chặt chẽ hơn thì trường hợp $b \neq 0$ thì *bx* + *c* mới được gọi là đa thức bậc nhất.

[6] Nếu đã xác định là lưu đồ con thì người ta thường không vẽ nút Start và End mà lưu đồ bắt đầu từ một cạnh vào và kết thúc với một cạnh ra.

[7] Hai đường gạch gạch màu đỏ trên Hình 2.3.1.

Hình 2.3.3 – Lưu đồ chương trình tính tổng



Trong cách vẽ lưu đồ trên lưu ý là đúng ra ta phải vẽ đuôi mũi tên của cạnh Sai (từ nút lựa chọn "*x* = 0 ?") đâm vào nút "Nhập *x*" nhưng để dễ nhìn hơn ta đã vẽ nó nhích lên trên và đâm vào cạnh đi xuống nút "Nhập *x*"[8]. Ta cũng đã viết mã C cho lưu đồ trên ở Bài 1.5 (Mã 1.5.5) nhưng ta sẽ học cách chuyển lưu đồ trên sang mã C một cách hệ thống trong bài sau nữa.

Bạn hãy chạy lưu đồ trên với các giá trị nhập cho *x* là 1 2 3 0. Khi chạy bạn thấy rằng các nút *B*, *C*, *D* có thể được thực thi nhiều lần. Nhìn trên lưu đồ ta thấy từ nút *B*, đến nút *C*, đến nút quyết định *D*, sau đó theo cạnh Sai của nút *D* về lại nút *B*. Như vậy có đường đi từ nút *B* đến nút *B*, ta gọi đây là một *chu trình* hay một *vòng lặp*. Từ nút *A* ta vào nút *B* là một nút trong vòng lặp *B-C-D-B* nên ta gọi *A* là *nút vào của vòng lặp*. Từ nút quyết định *D* đi theo cạnh Đúng ta ra khỏi vòng lặp nên *D* gọi là *nút thoát của vòng lặp*. Một vòng lặp có thể có nhiều nút vào và nút ra. Nút vào quyết định ta có rơi vào vòng lặp hay không. Ở trên vì nút vào *A* có thể đến được từ nút Start và *A* không là nút quyết định nên ta chắc chắn rơi vào vòng lặp. Khi đó nút ra quyết định

[8] Việc này không làm thay đổi lưu đồ.

ta có rời khỏi được vòng lặp hay không. Nếu ta không rời khỏi được vòng lặp thì ta sẽ chạy mãi trong vòng lặp mà không đến được nút End, do đó không dừng việc thực thi được, ta nói rằng ta bị *lặp vô tận* và lưu đồ bị *treo*[9]. Ở trên nút ra *D* là nút điều kiện mà cạnh ra có thể xảy ra (do $x = 0$ là đúng nếu người dùng nhập *x* là 0) nên vòng lặp trên không phải là lặp vô tận nên lưu đồ trên không bị treo.

Khi lưu đồ có vòng lặp thì việc phân tích lưu đồ là khó. Chẳng hạn số cách thực hiện tùy thuộc vào số cách mà vòng lặp được thực thi, đó là số lần mà vòng lặp được thực thi. Chẳng hạn ở trên, khả năng thực thi thứ nhất là khi người dùng nhập 0 ở lần nhập *x* đầu tiên, ứng với số lần thực hiện của vòng lặp là 1. Khả năng thứ hai là khi người dùng nhập 0 ở lần nhập *x* thứ hai, ứng với số lần thực hiện của vòng lặp là 2. … Như vậy có vô số khả năng thực thi. Hơn nữa khi có vòng lặp nghĩa là có nguy cơ bị lặp vô tận. Ở trên ta không bị lặp vô tận như đã phân tích nhưng trong những trường hợp phức tạp hơn thì việc xác định vòng lặp có bị lặp vô tận hay không là (rất) khó.

Ta có thể xem việc viết mã C chính là *cài đặt lưu đồ thành mã C*, nghĩa là ta dùng ngôn ngữ C (ngôn ngữ văn bản) để mô tả lưu đồ (ngôn ngữ hình ảnh, kí hiệu). Điều quan trọng là mô tả các công việc của các nút và thứ tự thực hiện (các cạnh của lưu đồ). C có các loại lệnh khác nhau cho các dạng lưu đồ khác nhau để mô tả các công việc khác nhau.

Lệnh đơn giản nhất của C là lệnh đơn (lệnh biểu thức mà ta đã biết trong Bài 2.2). Lệnh này mô tả các công việc đơn giản nhất, tương ứng chính là các nút của lưu đồ. C không có lệnh đặc biệt cho việc nhập xuất, chúng đều là lời gọi đến các hàm hỗ trợ nhập xuất. Vì *các lời gọi hàm đều là các biểu thức* nên các lệnh nhập xuất trong C thực ra là các lệnh đơn với ý nghĩa đặc biệt:

- *Nút nhập*: các hàm `scanf`, `gets`, … Cũng lưu ý là trong lưu đồ ta không để ý đến việc khai báo biến (thậm chí không nêu rõ kiểu của biến) và không để ý đến thông báo nhập. Nhưng khi cài đặt nút này xuống C ta phải để ý đến các vấn đề này.
- *Nút xuất*: các hàm `printf`, `puts`, … Cũng lưu ý là ta cũng không để ý đến định dạng xuất ra trong lưu đồ như trong mã C.
- *Nút thao tác*: các loại lệnh biểu thức khác thì xem như là nút thao tác với nút hay gặp chính là lệnh gán giá trị một biểu thức nào đó cho một biến nào đó. Cũng vậy ta đã không để ý đến việc khai báo biến. Biến nằm ở vế phải lệnh gán thì phải xuất hiện trước đó trong lưu đồ còn biến bên vế trái nếu xuất hiện lần đầu thì phải được khai báo khi ta viết thành mã C.

Thật ra lệnh đơn giản nhất của C là *lệnh rỗng* mà ta có thể xem là lệnh đơn đặc biệt. Cú pháp: `;` (chỉ gồm dấu `;`). Lệnh này mô tả *công việc rỗng*

---

[9] Khi viết thành chương trình thì ta nói rằng *chương trình bị treo*.

(không làm gì cả)[10]. Đó có thể xem là nút thao tác rỗng. Chắc bạn nghĩ rằng ta cần gì lệnh này? Bạn sẽ thấy một số trường hợp mà nó được dùng rất hay.

Bạn đã thấy việc ghép các lưu đồ để mô tả công việc lớn gồm nhiều công việc nhỏ nối tiếp nhau theo thứ tự thực hiện. C cũng cho phép ghép nhiều lệnh vào thành một lệnh lớn hơn đó là *lệnh ghép* với cú pháp:

```
{
    <các lệnh con theo thứ tự>
}
```

Do được bọc trong cặp dấu đóng mở ngoặc nhọn nên lệnh ghép còn được gọi là *khối*. Một khối đặc biệt là *khối rỗng* `{}` cũng được phép trong C. Thứ tự liệt kê các lệnh con sẽ là thứ tự thực thi. Nói cách khác các cạnh từ nút tương ứng với lệnh ở trước đến nút tương ứng với lệnh sau đó được tự động thêm vào. Cũng lưu ý, để tránh rườm rà ta có thể gom các lệnh đơn kế tiếp nhau (không phải là lệnh nhập xuất) vào trong một nút thao tác thay vì để mỗi lệnh trong một nút thao tác. Hình (a) dưới đây là một khối lệnh C[11] và Hình (b) là lưu đồ tương ứng của khối đó.

Hình 2.3.4 – Ví dụ về lưu đồ của khối

Hình (a)

```
{
    double r, c, s;

    printf("r = ? ");
    scanf("%lf", &r);


    c = 2 * 3.1416 * r;
    s = 3.1416 * r * r;


    printf("Chu vi: %lf.\nDien tich:
                %lf.\n", c, s);
}
```

Hình (b)

Nhập *r*

$c = 2\pi r$
$s = \pi r^2$

Xuất *c*, *s*

Ngoài ra C có các lệnh, có cú pháp rất đơn giản, gọi là *lệnh nhảy* được dùng với ý nghĩa đặc biệt liên quan đến luồng điều khiển (tức là các cạnh của lưu đồ) mà ta sẽ tìm hiểu sau[12]. Các lệnh khác của C gọi là *lệnh phức* vì

[10] Nếu ai ngạc nhiên khi có một dạng việc là "không làm gì cả" thì hãy cứ "làm thinh" nhé!:).

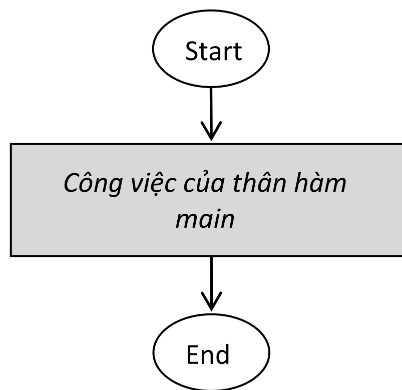
[11] Khối này là thân của hàm main trong chương trình ở Bài 1.3 (Mã 1.3.1).

[12] Lệnh `return` mà ta hay gặp ở cuối thân hàm `main` là một loại lệnh nhảy.

về mặt cú pháp nó có chứa các lệnh nhỏ hơn bên trong mà về ngữ nghĩa chúng cho phép tạo các lưu đồ với các hướng thực thi khác nhau (*lệnh lựa chọn*) và lưu đồ với các vòng lặp (*lệnh lặp*).

Nút Start và End mô tả thế nào trong C? Thân hàm `main` chính là một khối. Chương trình bắt đầu chạy từ thân hàm `main` và kết thúc khi hàm `main` kết thúc. Nói cách khác ta có lưu đồ cho chương trình như hình sau. Công việc của thân hàm `main` được để như là một hộp đen và bằng cách phân giải hộp đen này (ứng với lưu đồ của khối thân hàm `main`) thì ta có lưu đồ chạy được của chương trình.

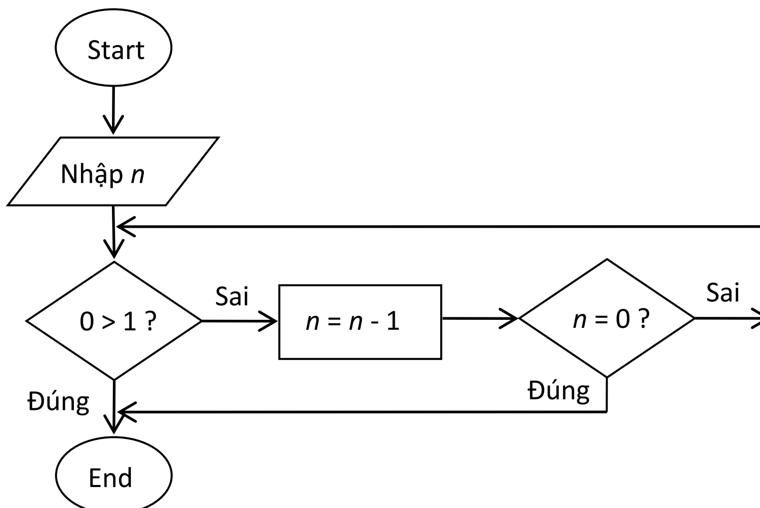
Hình 2.3.5 – Lưu đồ của chương trình



BÀI TẬP

Bt 2.3.1 Cho biết lưu đồ sau sẽ dừng/không dừng trong trường hợp nào?

Hình 2.3.6 – Lưu đồ của Bt 2.3.1



Bt 2.3.2 Vẽ lưu đồ cho chương trình "Calculator đơn giản" (Mã 1.2.2).

Bt 2.3.3 Vẽ lưu đồ cho công việc tính giai thừa một số nguyên dương (tương tự Mã 1.5.4).

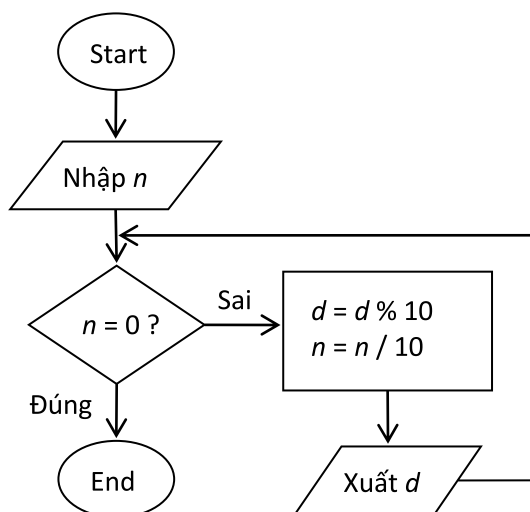
Bt 2.3.4 Vẽ lưu đồ cho việc tính và phân loại chỉ số BMI (Bài tập 1.5.3).

Bt 2.3.5 Vẽ lưu đồ cho công việc tính căn bậc hai của một số dương (Mở rộng 1.5).

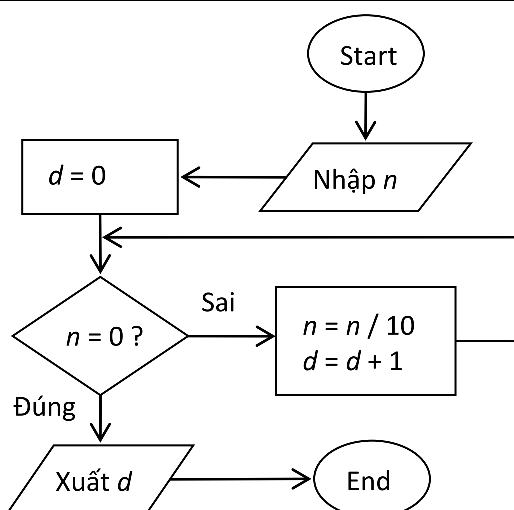
Bt 2.3.6 Với từng lưu đồ ở hình sau, cho biết kết quả xuất ra khi chạy lưu đồ với giá trị nhập cho *n* là 12345, cho biết công việc của lưu đồ và viết chương trình C tương ứng cho lưu đồ.

Hình 2.3.7 – Các lưu đồ của Bài tập 2.3.6

Hình (a)



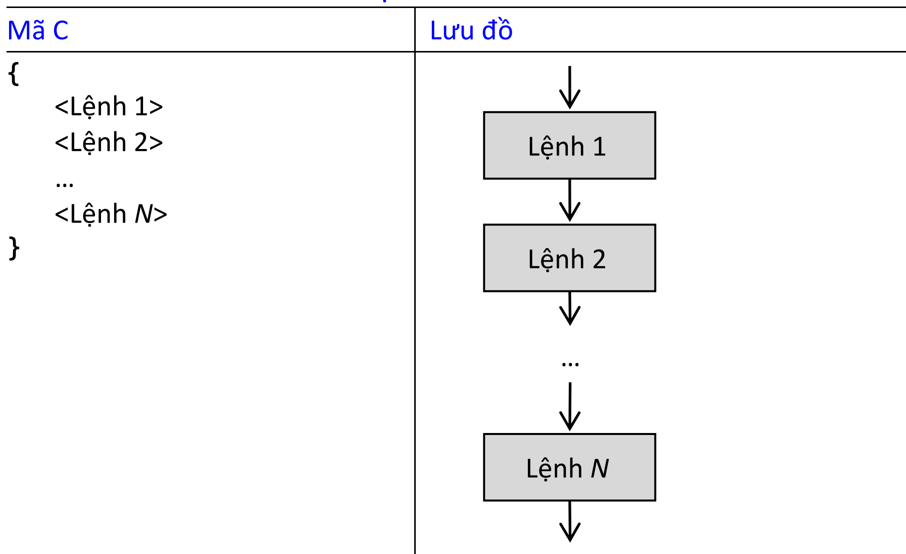
Hình (b)



BÀI 2.4

Với *lệnh ghép* (khối lệnh) ta đã có thể mô tả các công việc lớn gồm nhiều bước là các công việc nhỏ hơn (lưu đồ gồm nhiều nút).

Hình 2.4.1 – Lưu đồ của khối lệnh

Mã C	Lưu đồ
{ <Lệnh 1> <Lệnh 2> … <Lệnh *N*> }	

Ta đã dùng hộp đen cho các lệnh để chỉ rằng ta chưa phân giải cụ thể: nó có thể là bất kì lệnh gì. Do lưu đồ không có nút lựa chọn nên ta chỉ có một luồng (hay một hướng, một cách) thực thi: *thực thi tuần tự* từ <Lệnh 1>, đến <Lệnh 2>, …, đến <Lệnh *N*>. Để có thể mô tả công việc gồm nhiều hướng (tương ứng là lưu đồ gồm nhiều nhánh với các nút lựa chọn), C cung cấp các *lệnh lựa chọn* (hay *rẽ nhánh*). Có hai lệnh lựa chọn trong C là `if` và `switch` (đặt tên theo từ khóa). Lệnh `switch` sẽ được tìm hiểu sau còn lệnh `if` thì có hai dạng: `if`-đủ (`if-else`) và `if`-khuyết. Cú pháp của hai dạng này và lưu đồ tương ứng được mô tả ở Hình 2.4.2. Trong đó *E* là biểu thức luận lý gọi là *điều kiện* của `if`, <Lệnh *A*> là một lệnh (bất kì) gọi là *thân* của `if`, và <Lệnh *B*> là một lệnh (bất kì) gọi là thân của `else`, còn `if`, `else` là các từ khóa. Bạn hãy xem lại Bài 2.1 để nắm biểu thức luận lý.

Lưu ý là `if`-khuyết được xem là trường hợp đặc biệt của `if`-đủ nếu <Lệnh *B*> là lệnh rỗng, nghĩa là C chỉ cần cung cấp một dạng `if` là `if-else` mà không cần dạng kia. Ngược lại, `if`-đủ cũng có thể được mô tả bằng `if`-khuyết như Hình 2.4.3.

Hình 2.4.2 – Lưu đồ của lệnh `if`-đủ (`if-else`) và `if`-khuyết

Mã C	Lưu đồ
if(<Điều kiện *E*>) <Lệnh *A*> **else** <Lệnh *B*>	Đúng Đk *E* ? Sai Lệnh *A* Lệnh *B*
if(<Điều kiện *E*>) <Lệnh *A*>	Đúng Đk *E* ? Sai Lệnh *A*

Hình 2.4.3 – Lưu đồ cho thấy `if`-đủ cũng có thể mô tả bằng `if`-khuyết

Mã C	Lưu đồ
if(<*E*>) <Lệnh *A*> **if**(!<*E*>) <Lệnh *B*>	Đúng *E* ? Lệnh *A* Sai Đúng (*E* sai) !*E* ? Lệnh *B* Sai (*E* đúng)

Quan sát lưu đồ Hình 2.4.3 ta thấy khi *E* đúng thì <Lệnh *A*> được thực thi và do `!`*E* sai (phủ định *E* sai khi *E* đúng) nên <Lệnh *B*> không được thực thi. Ngược lại, khi *E* sai thì <Lệnh *A*> không được thực thi và do `!`*E* đúng (phủ định *E* đúng khi *E* sai) nên <Lệnh *B*> được thực thi. Như vậy có thể nói rằng lưu đồ trên tương đương với lưu đồ của `if`-đủ[1]. Điều đó cũng có nghĩa là C chỉ cần cung cấp dạng `if`-khuyết là được mà không cần dạng kia. Tuy nhiên việc C cung cấp cả 2 dạng `if` là tốt: cùng một nội dung có thể được thể hiện (viết, mô tả) theo nhiều cách khác nhau.

Lưu ý là điều ngược lại: cùng một mô tả nhưng có nhiều nội dung (cách hiểu) khác nhau thì phải tuyệt đối tránh. Sự nhập nhằng, mơ hồ như vậy cũng thường xảy ra trong ngôn ngữ tự nhiên nhưng không được phép trong các ngôn ngữ lập trình như C bởi vì compiler sẽ không biết biên dịch như thế nào khi gặp mã C như vậy. Chẳng hạn một trường hợp nhập nhằng hay gặp trong lệnh `if` là trường hợp “`else` *thòng*”:

```
if(<E1>)
if(<E2>)
<S1>
else
<S2>
```

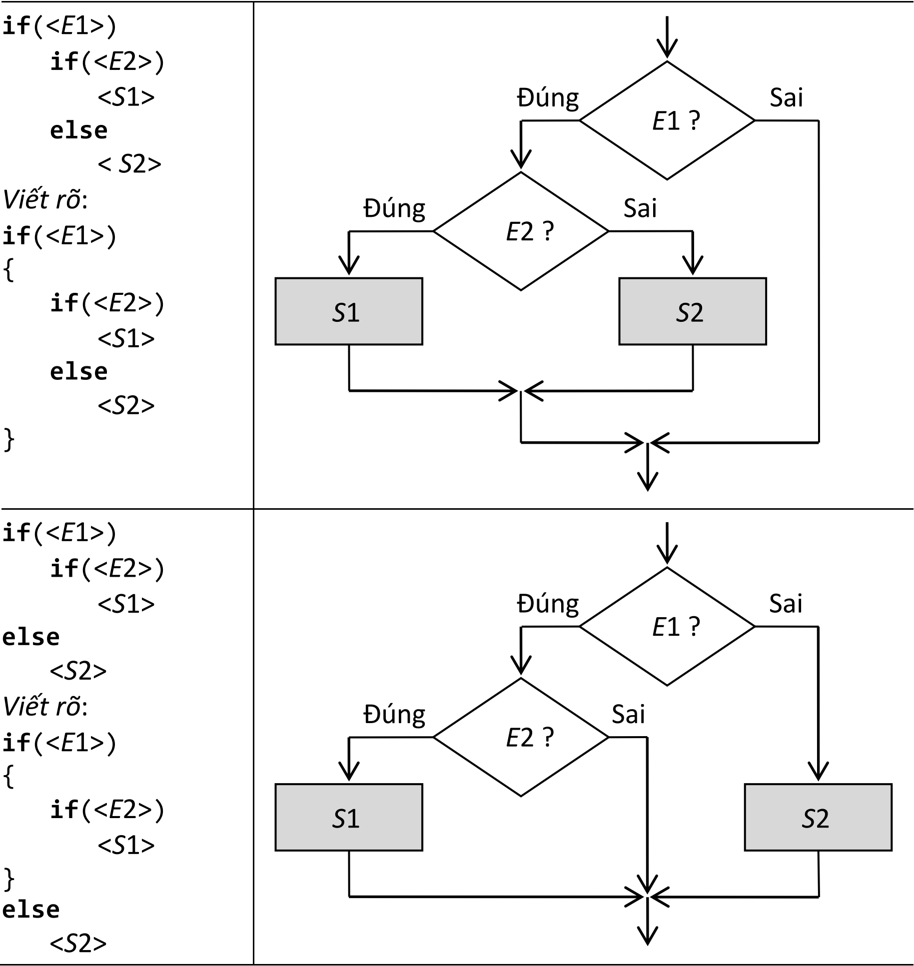
Mã C trên có hai cách hiểu với cách viết mã và lưu đồ tương ứng như Hình 2.4.4. Lưu ý là compiler không để ý đến việc định dạng mã (cách ra, xuống dòng) nên việc định dạng không quyết định cách hiểu của compiler. Ở đây, cả hai cách hiểu đều đúng cú pháp của `if`-khuyết và `if`-đủ (lưu ý là thân của `if` cũng như `else` có thể là bất kì lệnh nào khác, bao gồm cả lệnh `if`). Tuy nhiên compiler sẽ chọn cách hiểu thứ nhất dựa trên qui tắc bất thành văn[2] là “`else` đi với `if` gần nhất”. Nếu ta cần C hiểu theo cách thứ hai (trường hợp chương trình của ta như vậy) thì ta dùng khối để *phân giải nhập nhằng* (dùng cặp ngoặc nhọn `{}` như đã dùng cặp ngoặc tròn `()` để xác định độ ưu tiên trong biểu thức) như cách viết rõ đã chỉ ra. Khi đó, theo cú pháp, chỉ có duy nhất một cách hiểu như ta đã viết vì `else` không thể đi với `if` bên trong khối được.

Hình 2.4.4 – Trường hợp nhập nhằng của lệnh `if` (“`else` thòng”)

Mã C	Lưu đồ

[1] Thực ra hai lưu đồ không thật sự tương đương do trong `if` đủ thì *E* được lượng giá chỉ một lần còn trong lưu đồ dùng 2 `if`-khuyết thì *E* được lượng giá hai lần, điều này có thể khác biệt khi *E* có hiệu ứng lề.

[2] Nhưng cần phải nhớ.



Một cách khác để cho thấy các hướng thực thi khác nhau dựa trên các điều kiện nào đó là dùng *bảng quyết định*. Chẳng hạn bảng quyết định của 2 cách hiểu trên cho thấy rõ chúng khác nhau thế nào.

Bảng 2.4.1 – Bảng quyết định của hai cách hiểu "`else` thòng"

Bảng quyết định của cách 1		
Trường hợp		Thực thi
*E*1 đúng	*E*2 đúng	*S*1
	*E*2 sai	*S*2
*E*1 sai	*E*2 đúng	-
	*E*2 sai	-

Bảng quyết định của cách 2		
Trường hợp		Thực thi
*E*1 đúng	*E*2 đúng	*S*1
	*E*2 sai	-
*E*1 sai	*E*2 đúng	*S*2
	*E*2 sai	*S*2

Bảng trên cho thấy trong trường hợp nào thì thực hiện việc gì, chẳng hạn "Bảng quyết định của cách hiểu 1" cho thấy trường hợp *E*1 đúng và *E*2 đúng thì thực hiện *S*1 còn *E*1 đúng và *E*2 sai thì thực hiện *S*2 còn *E*1 sai (bất kể *E*2

đúng hay sai) thì không làm gì cả (kí hiệu bằng dấu -). "Bảng quyết định của cách hiểu 2" thì khác, chẳng hạn trường hợp *E*1 đúng và *E*2 sai thì không làm gì cả ở cách hiểu 2.

Mặc dù C chỉ có hai dạng `if` là `if`-đủ và `if`-khuyết nhưng vì thân `if` cũng như `else` là lệnh có dạng bất kì nên bằng cách tùy biến điều kiện của `if`, thân `if`, thân `else` mà ta có vô số cách vận dụng khác nhau không kể siết!:) Chẳng hạn về lý thuyết thì điều kiện của `if` là nhị phân, tức là có hai trường hợp thực hiện tùy theo điều kiện rẽ nhánh là đúng hay sai nhưng ta cũng có thể mô tả trường hợp có nhiều hướng rẽ nhánh bằng cách dùng "`if` *lồng*" (thân của `if` hay `else` lại là lệnh `if`) mà dạng phổ biến nhất là là "`if` *tầng*".

Hình 2.4.5 – `if` tầng

<table>
<tr><th>Hình (a)</th><th>Hình (b)</th><th>Hình (c)</th><th>Hình (d)</th></tr>
<tr>
<td><pre>if(<E1>)
    <S1>
else
{
    if(<E2>)
        <S2>
    else
        <S3>
}</pre></td>
<td><pre>if(<E1>)
    <S1>
else if(<E2>)
    <S2>
else
    <S3></pre></td>
<td><pre>if(<E1>)
    <S1>
if(!<E1>
       &&<E2>)
    <S2>
if(!<E1>
       &&!<E2>)
    <S3></pre></td>
<td><table>
<tr><th colspan="3">Bảng quyết định</th></tr>
<tr><th colspan="2">Trường hợp</th><th>Thực thi</th></tr>
<tr><td>E1 đ</td><td>-</td><td>S1</td></tr>
<tr><td rowspan="2">E1 s</td><td>E2 đ</td><td>S2</td></tr>
<tr><td>E2 s</td><td>S3</td></tr>
</table></td>
</tr>
</table>

Ở trên thì có 3 trường hợp thực thi:

- *S*1 được thực hiện nếu *E*1 đúng (mà không cần biết *E*2 thế nào, thể hiện bằng dấu - trong Bảng quyết định như Hình (d) trên).
- *S*2 được thực hiện nếu *E*1 sai và *E*2 đúng.
- *S*3 được thực hiện nếu cả *E*1 và *E*2 đều sai.

Do các trường hợp trên là rời nhau hay loại trừ nhau (không thể cùng đúng) nên ta có thể viết lại bằng cách rõ ràng hơn nhưng dài dòng hơn như ở Hình (c). Mặc dù tôi đã khuyến khích các bạn nên viết rõ ràng hơn nhưng viết ngắn gọn như Hình (b) cũng tốt (trong trường hợp này thì ngắn gọn nhưng cũng khá rõ ràng và nhất là nhiều người đều viết như vậy). Cách viết ở Hình (b) gọi là `if` tầng mà lí do đó là bạn có thể nối nhiều `if else` vào nhau và S_n sẽ được thực hiện khi E_1, E_2, ..., E_{n-1} sai còn E_n đúng. Còn lệnh đi với `else` sau cùng sẽ chỉ được thực hiện khi tất cả các *E* đều sai. Cách viết rõ hơn của `if` tầng, như ở Hình (a) cho thấy, thực ra nó chính là một trường hợp của `if` lồng (thân của `else` là lệnh `if-else`).

Trong Bài tập 1.5.3 của Bài 1.5 bạn đã thấy cách phân loại chỉ số BMI. Bảng phân loại đó có thể được viết lại bằng Bảng quyết định sau. Có lẽ bạn đã viết mã C bằng cách thông thường (nhiều `if`-khuyết tuần tự như Hình (c) trên). Bạn hãy vẽ lưu đồ và viết lại mã bằng cách dùng `if` tầng nhé.

Bảng 2.4.2 – Bảng quyết định phân loại chỉ số BMI

<table>
<tr><th colspan="3">Trường hợp</th><th>Phân loại</th></tr>
<tr><td>BMI < 18.5</td><td colspan="2">-</td><td>Gầy</td></tr>
<tr><td rowspan="3">BMI ≥ 18.5</td><td>BMI < 25</td><td>-</td><td>Bình thường</td></tr>
<tr><td rowspan="2">BMI ≥ 25</td><td>BMI < 30</td><td>Béo</td></tr>
<tr><td>BMI ≥ 30</td><td>Béo phì</td></tr>
</table>

Chắc chắn bạn đã thắc mắc tại sao lại cần lệnh đơn rỗng (lệnh chỉ có dấu `;`). Sau đây là một lí do cho thấy là … không nên có lệnh rỗng:-). Những người mới viết mã C thường hay mắc phải một lỗi là dư dấu chấm phẩy trong lệnh `if` như sau:

```
if(<E>);
    <Lệnh S>
```

Theo thói quen kết thúc lệnh đơn bằng dấu `;` ta đã vô tình viết dư dấu `;` như trên nhưng thật ra ý định của ta là:

```
if(<E>)
    <Lệnh S>
```

Tuy nhiên compiler vẫn chấp nhận cách viết dư dấu `;` vì đoạn mã đó vẫn đúng cú pháp (đó không phải là lỗi biên dịch). Dấu `;` được xem là một lệnh đơn hợp lệ và như vậy thân của `if` là lệnh đơn `;` chứ không phải là <Lệnh *S*>. Còn <Lệnh *S*> là lệnh kế tiếp của lệnh `if`. Tức là compiler sẽ hiểu như sau.

Hình 2.4.6 – Mã C và lưu đồ của lỗi "`if` dư dấu `;`"

Hình (a)	Hình (b)	Hình (c)
if(<*E*>) **;** <Lệnh *S*>	<*E*>**;** <Lệnh *S*>	Đúng *E* ? Sai Lệnh *S*

Hình (a) ở trên là cách định dạng mã đúng với cách hiểu của C còn Hình (c) là lưu đồ thực thi tương ứng. Từ đó ta thấy điều kiện <*E*> sẽ được lượng

giá và nếu <*E*> đúng thì thực hiện lệnh rỗng (mà nghĩa là không làm gì cả). Sau đó dù <*E*> đúng hay sai thì <Lệnh *S*> đều được thực hiện (<Lệnh *S*> là lệnh kế tiếp lệnh `if` chứ không phải là thân `if`). Như vậy thực ra là lượng giá <*E*> sau đó là thực hiện <Lệnh *S*> mà điều này tương đương với mã trong Hình (b). Hơn nữa nếu <*E*> không có hiệu ứng lề thì việc lượng giá <*E*> là không cần thiết nên mã tương đương với thực hiện <Lệnh *S*>.

Lỗi dư dấu `;` như trên, như vậy, là lỗi logic và do đó ta cần cực kì thận trọng vì nếu mắc lỗi thì ta cũng không được cảnh báo gì. Ngược lại nếu viết dư dấu `;` trong `if`-đủ như hình sau (mã trong Hình (b) là định dạng lại mã của Hình (a) để tránh bị "đánh lừa thị giác":)).

Hình 2.4.7 – Mã C và lưu đồ của lỗi "dư dấu `;` trong `if`-đủ"

Hình (a)	Hình (b)
if(<*E*>); <Lệnh *A*> **else** <Lệnh *B*>	**if**(<*E*>) ; <Lệnh *A*> **else** <Lệnh *B*>

thì nếu <Lệnh *A*> không phải là lệnh `if` thì đây sẽ là lỗi cú pháp do từ khóa `else` đi một mình mà không khớp với từ khóa `if` nào trước đó để tạo lệnh `if` đủ hợp lệ. Tuy nhiên nếu <Lệnh *A*> lại là lệnh `if`-khuyết thì đó lại có thể là đoạn mã hợp lệ và do đó đó là lỗi logic. Chẳng hạn, bạn hãy phân tích đoạn mã sau và vẽ lưu đồ để thấy cách mà C hiểu nhé (cũng vậy, mã trong Hình (b) là định dạng lại mã của Hình (a) để "dễ nhìn thấy hơn").

Hình 2.4.8 – Một trường hợp "dư dấu `;` trong `if`-đủ" không bị lỗi cú pháp

Hình (a)	Hình (b)
if(<*E*>); **if**(<F>) <Lệnh *C*> **else** <Lệnh *B*>	**if**(<*E*>) ; **if**(<F>) <Lệnh *C*> **else** <Lệnh *B*>

Tiện thể, một lưu ý khác về lỗi hay gặp là trong điều kiện của `if` (cũng như điều kiện của các lệnh lặp sẽ học sau) thường gõ sai toán tử so sánh bằng (==) với toán tử gán (=). Ví dụ:

```
if(x = 1)
    printf("x la mot");
else
    printf("x khac mot");
```

Đoạn mã trên luôn xuất ra: `x la mot`, cho dù biến `x` có chứa giá trị gì đi nữa. Bạn coi lại phần biểu thức trong Bài 2.1 và Bài 2.2 nhé.

BÀI TẬP

Bt 2.4.1 Vẽ lưu đồ và viết chương trình xếp loại học lực của sinh viên từ điểm trung bình theo bảng sau đây[3]:

- Từ 5 trở lên là Đạt. Hơn nữa:
  - Từ 9 đến 10: Xuất sắc.
  - Từ 8 đến cận 9: Giỏi.
  - Từ 7 đến cận 8: Khá.
  - Từ 6 đến cận 7: Trung bình – Khá.
  - Từ 5 đến cận 6: Trung bình.
- Dưới 5 là Không đạt. Hơn nữa:
  - Từ 4 đến cận 5: Yếu.
  - Dưới 4: Kém.

Bt 2.4.2 Vẽ lưu đồ và viết chương trình tìm số lớn nhất, nhỏ nhất trong 3 số.

Bt 2.4.3 Viết chương trình tìm số lớn nhất, nhỏ nhất trong 4 số.

Bt 2.4.4 Vẽ lưu đồ và viết chương trình cho nhập 3 số, xuất ra theo thứ tự tăng dần.

Bt 2.4.5 Viết chương trình cho nhập 3 số, xuất ra theo thứ tự tăng dần.

Bt 2.4.6 Thời điểm trong ngày được xác định bởi giờ, phút, giây theo hệ 24 giờ, nghĩa là từ 0 giờ : 0 phút : 0 giây đến 23 giờ : 59 phút : 59 giây. Viết chương trình cho nhập 2 thời điểm, kiểm tra tính hợp lệ, cho biết thời điểm nào trước thời điểm nào (hay trùng nhau) và cho biết tổng số giây trôi qua giữa hai thời điểm đó.

Bt 2.4.7 Cho biết các đoạn mã sau có chạy được không (có lỗi biên dịch không)? Nếu có hãy cho biết kết quả xuất ra, nếu không hãy chỉ ra chổ lỗi:

```
if(1 < 2 > 3 < 4)
    printf("1");;
printf("2");
```

```
if(1)
printf("1");
else;
printf("0");
```

```
if(3 = 2);
{
}
else
    printf("1");
```

[3] Theo “Qui chế học vụ” năm 2016 của Trường ĐH KHTN, Tp. HCM.

BÀI 2.5

Với các lệnh ghép và lệnh lựa chọn ta đã có thể cài đặt các lưu đồ phức tạp gồm nhiều nút với nhiều hướng thực hiện. Tuy nhiên ta vẫn chưa thể cài đặt được các lưu đồ có vòng lặp (tương ứng với các công việc có lặp) do các cạnh đều hướng từ trên xuống[1]. Để có thể cài đặt các lưu đồ có vòng lặp, C cung cấp các *lệnh lặp*. Các lệnh lặp trong C là `while`, `do-while` và `for`.

Lệnh `while` là lệnh lặp đơn giản nhất có cú pháp và lưu đồ như sau.

Hình 2.5.1 – Lưu đồ của lệnh `while` (so sánh với `if`-khuyết)

Mã C	Lưu đồ
`if`(<*E*>) <*S*>	*E* ? Đúng Sai *S*
`while`(<*E*>) <*S*>	*E* ? Đúng Sai *S*

Về mặt cú pháp thì lệnh `while` chỉ khác lệnh `if`-khuyết ở từ khóa `while` thay cho `if`, còn *E* vẫn là biểu thức luận lý và được gọi là điều kiện của `while` (hay điều kiện lặp); *S* vẫn là lệnh bất kì và được gọi là thân của `while`. Khác biệt quan trọng là ngữ nghĩa của hai lệnh này. Như lưu đồ cho thấy, trong `while` thì từ nút *S* có mũi tên ngược lên nút *E*. Điều đó có nghĩa là sau

[1] Người ta thường vẽ lưu đồ từ trên xuống và không vẽ mũi tên với cạnh đi từ trên xuống.

khi lượng giá *E* mà *E* đúng thì thực thi *S*, sau đó lại thực hiện lại lệnh `while` (nghĩa là lại lượng giá *E* mà *E* đúng thì thực thi *S*, sau đó lại thực hiện lại lệnh `while`, ...). Với lệnh `if` thì sau khi lượng giá *E* mà *E* đúng thì thực thi *S* và kết thúc lệnh `if` (và tiếp tục thực thi lệnh kế tiếp nếu có). Nếu *E* sai thì lệnh `while` kết thúc giống như `if`. Quan sát trong lưu đồ của `while` ta thấy có một vòng lặp hay chu trình (một đường đi bắt đầu từ một nút và về lại nút đó) là: *E* (Đúng) $\to$ *S* $\to$ *E* (cũng có thể gọi đó là chu trình *S* $\to$ *E* (Đúng) $\to$ *S*). Chính vòng lặp này giúp *S* được thực thi lặp đi lặp lại chừng nào *E* còn đúng. Gọi số lần lặp *N* là số lần vòng lặp *E* (Đúng) $\to$ *S* $\to$ *E* được thực thi. Ta thấy *N* cũng chính là số lần *S* được thực thi và $N \geq 0$. Như vậy điều khác biệt căn bản về ngữ nghĩa của `if`-khuyết với `while` là số lần thực thi thân của `if` là ≤ 1 (tức là 0 hoặc 1 lần) còn số lần thực thi thân của `while` là ≥ 0 (tức là 0, 1, 2, ... lần).

Lệnh lặp `do-while` gần giống như `while`, có cú pháp và lưu đồ như sau.

Hình 2.5.2 – Lưu đồ của lệnh `do-while` (so sánh với `while`)

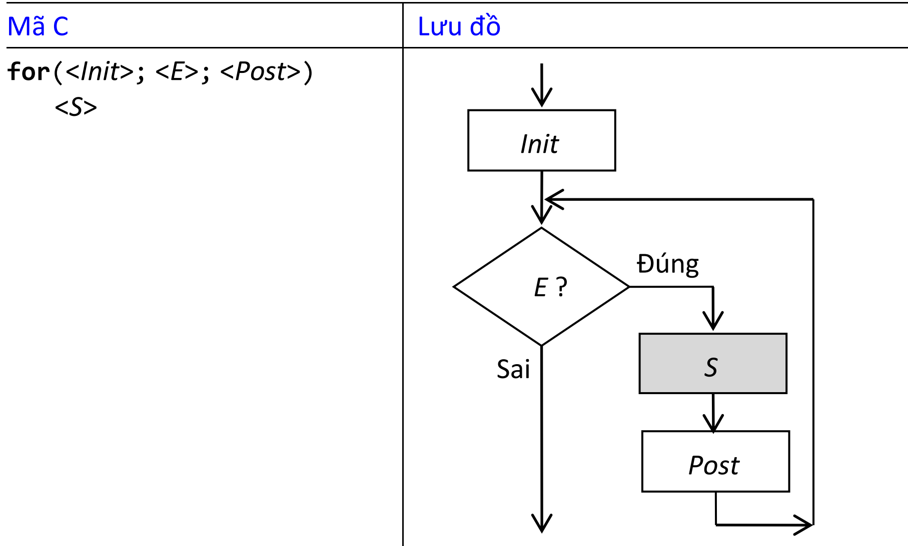
Mã C	Lưu đồ
`while`(<*E*>) <*S*>	*E* ? — Đúng → *S* → (quay lại *E* ?); Sai →
do <*S*> **`while`**(<*E*>);	*S* → *E* ? — Đúng → (quay lại *S*); Sai →

Về mặt cú pháp thì lệnh `do-while` dùng 2 từ khóa `do`, `while` và có dấu `;` kết thúc lệnh như hình trên. Về mặt ngữ nghĩa thì ở `do-while` thân lệnh *S* sẽ được thực thi trước rồi mới kiểm tra điều kiện *E*. Điều đó dẫn đến số lần

thực hiện *S* là ≥ 1. Bạn hãy tự so sánh 2 lưu đồ để thấy rõ khác biệt giữa lệnh `do-while` và `while` nhé.

Lệnh lặp cuối cùng, lệnh `for` thì có cú pháp và lưu đồ thực thi phức tạp hơn như sau.

Hình 2.5.3 – Lưu đồ của lệnh `for`



E vẫn là biểu thức luận lý và được gọi là điều kiện của `for` (hay *điều kiện lặp*). *Init* được gọi là *tiền lệnh* của `for`, nó được thực thi một lần duy nhất trước khi kiểm tra *E* (tức là trước khi lặp), do đó nó cũng được gọi là *lệnh khởi động*. *S* là lệnh bất kì và được gọi là *thân* của `for`. *Post* được gọi là *hậu lệnh* của `for`, nó được thực thi ngay sau khi thực thi thân lệnh *S*. Lưu ý là trong khi *S* có thể là một lệnh có dạng bất kì thì *Init* và *Post* chỉ có thể là các lệnh đơn[2]. Hơn nữa *Init* có thể là cấu trúc khai báo biến và khởi tạo như:

```
int i = 1
```

Cũng như với lệnh rẽ nhánh, thật ra ta chỉ cần một trong 3 dạng lặp trên là đủ vì các dạng khác có thể đưa về lẫn nhau. Ví dụ lệnh `for`:

```
for(<Init>; <E>; <Post>)
    <S>
```

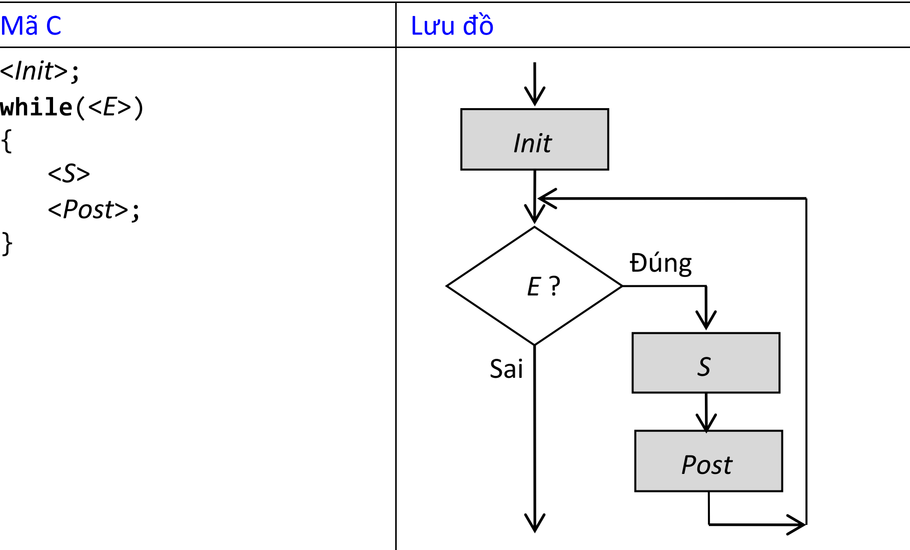
có thể được viết bằng lệnh `while` như hình sau. Ta thấy lưu đồ của cách viết thứ 2 (không dùng `for`) hoàn toàn giống lưu đồ của lệnh `for`. Đúng ra là cách viết thứ 2 linh hoạt hơn cách dùng lệnh `for` vì với cách viết thứ 2 thì

---

[2] Thực ra thì về cú pháp chúng là các biểu thức nhưng về ngữ nghĩa ta nên hiểu chúng như là lệnh, tức là biểu thức có hiệu ứng lề.

Init và *Post* có thể là những lệnh bất kì chứ không hạn chế như trong lệnh `for`[3].

Hình 2.5.4 – Lưu đồ của lệnh `for` viết bằng `while`



Bạn hãy suy ngẫm để nắm rõ nhé. C có 3 dạng lệnh lặp và như vậy có 6 cặp chuyển đổi. Tôi đã chỉ ra cho cặp `for` ⇒ `while`. Bạn có thể đưa ra cách chuyển đổi cho 5 cặp còn lại không?

Mặc dù các dạng lệnh lặp có thể thay thế cho nhau nhưng mỗi dạng lại tỏ ra tự nhiên với một số tình huống lặp nào đó. Chẳng hạn khi không biết trước số lần lặp và có thể không lặp lần nào thì ta hay dùng lệnh `while` còn khi có ít nhất một lần lặp thì ta hay dùng `do-while`. Ngược lại, nếu biết trước số lần lặp thì ta hay dùng lệnh `for` như khuôn mẫu "Làm *N* lần" trong Bài 1.5. Linh hoạt hơn, ta thường dùng `for` để lặp lại công việc nào đó theo biến `i` nguyên với `i` lần lượt nhận các giá trị cụ thể nào đó (khi đó `i` được gọi là *biến lặp*[4]). Chẳng hạn cách dùng `for` trong Hình 2.5.5 sẽ giúp lặp lại công việc *S* khi *i* lần lượt nhận các giá trị là 1, 2, …, *n*. Ở trên tôi đã kí hiệu *S*(*i*) để ám chỉ công việc *S* phụ thuộc vào giá trị của *i*. Ở lần lặp thứ nhất, *S* được thực thi với giá trị của *i* là 1, lần lặp thứ hai, *S* được thực thi với giá trị của *i* là 2, …, lần lặp cuối cùng, *S* được thực thi với *i* là *n*. Khi *i* tăng lên thành $n + 1$ thì điều kiện $i \le n$ không còn đúng nên lệnh `for` kết thúc (mà không lặp thêm *S*). Cũng rõ ràng là nếu biến `n` chứa giá trị nhỏ hơn 1 thì thân lệnh *S* không được thực thi lần nào, lệnh `for` kết thúc ngay khi kiểm tra điều kiện `i <= n`. Nếu bạn chưa thấy rõ thì hãy nhìn lưu đồ sẽ thấy rõ hơn nhé. Cũng

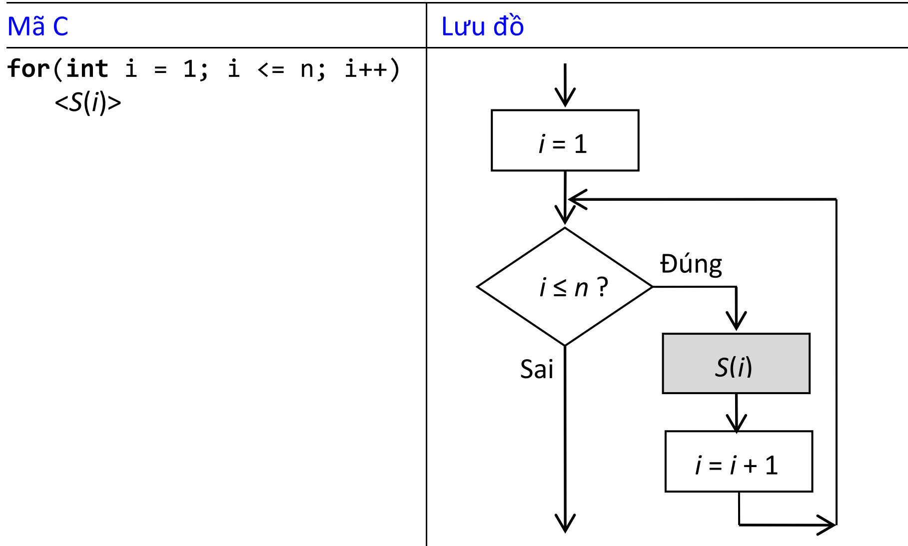
---

[3] Bạn có tinh ý nhận ra tôi đã mô tả điều đó bằng hộp đen cho *Init* và *Post* để nói rằng nó có thể là lệnh bất kì không?

[4] Hiển nhiên là ta có thể dùng tên biến khác như `j`, `k`, … làm biến lặp.

lưu ý là biến lặp `i` có thể được khai báo tại tiền lệnh của `for` hoặc có thể được khai báo bên ngoài và khi đó tiền lệnh chỉ là: `i = 1`[5]. Bạn xem lại Bài 1.5 để thấy cách ta vận dụng khuôn mẫu này để tính giai thừa của một số nguyên dương.

Hình 2.5.5 – Khuôn mẫu "Làm *N* lần" với `for`

Mã C	Lưu đồ
`for(int i = 1; i <= n; i++)` <*S*(*i*)>	

Ta cũng thường cải biến khuôn mẫu trên để lặp lại công việc *S*(*i*) khi *i* lần lượt nhận các giá trị: bắt đầu từ *a*, tăng dần mỗi đơn vị, cho đến *b* hay bắt đầu từ *b*, giảm dần mỗi 2 đơn vị, cho đến *a*, ... Chẳng hạn bạn hãy vận dụng lệnh `for` để viết chương trình xuất ra các số chẵn từ 100 giảm dần đến 50 thử nhé. Trong những khuôn mẫu này thì *Init* giúp ta đặt giá trị đầu tiên của *i* (chẳng hạn *i* = *b*), *Post* giúp ta điều chỉnh giá trị *i* dựa trên giá trị *i* trước đó (chẳng hạn *i* = *i* – 2, nghĩa là *i* sau sẽ giảm 2 đơn vị so với *i* trước) và *E* chỉ ra giá trị cuối của *i* (chẳng hạn *i* ≥ *b*).

Một điều rắc rối của lệnh `for` (và cũng là sự linh hoạt của nó) là cả 3 phần *Init*, *E*, *Post* đều có thể rỗng. Khi *Init*, *Post* rỗng thì nó được xem như lệnh rỗng (nghĩa là nút của nó được gỡ khỏi lưu đồ). Khi *E* rỗng thì nó được xem như là biểu thức 1 (nghĩa là biểu thức hằng có chân trị luôn là Đúng). Chẳng hạn lệnh `for(; <E>;)`<*S*> thì tương đương với lệnh `while(`<*E*>`)`<*S*>. Còn như lệnh `for(;;) printf("Hi");` sẽ lặp mãi không dừng việc xuất ra chuỗi `Hi`.

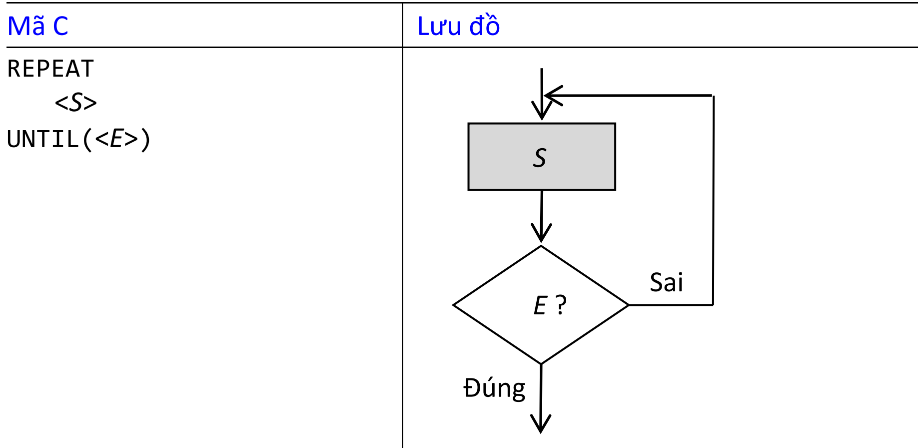
Một cấu trúc lặp cũng hay gặp mà C không có đó là cấu trúc "*Làm cho đến khi*" hay "*Repeat-until*"[6].

---

[5] Có khác biệt giữa 2 cách viết này mà ta sẽ thấy sau.

[6] Một vài ngôn ngữ lập trình khác, như Pascal, có cấu trúc này.

Hình 2.5.6 – Lưu đồ của cấu trúc REPEAT-UNTIL

Mã C	Lưu đồ
REPEAT <*S*> UNTIL(<*E*>)	

Với "Làm *S* cho đến khi *E*" ta thực hiện lặp lại việc *S* cho đến khi *E* đúng thì dừng. Việc thực hiện này được mô tả trong lưu đồ trên. Cấu trúc lặp của C gần gũi nhất với "Làm cho đến khi" chính là "Làm trong khi" hay `do-while`. Trong "Làm *S* trong khi *E*" ta thực hiện lặp lại việc *S* trong khi *E* đúng, nghĩa là lặp lại việc *S* cho đến khi *E* sai thì dừng. Mà *E* sai khi `!`*E* đúng nên "Làm *S* cho đến khi *E*" tương đương với "Làm *S* trong khi không *E*", nghĩa là ta có thể dùng lệnh `do-while` như sau:

```
do
    <S>
while(!<E>);
```

cho cấu trúc lặp "Làm *S* cho đến khi *E*". Bạn đối chiếu 2 lưu đồ để thấy rõ nhé.

MỞ RỘNG 2.5 – Lặp vô tận và nghi vấn 3*n* + 1

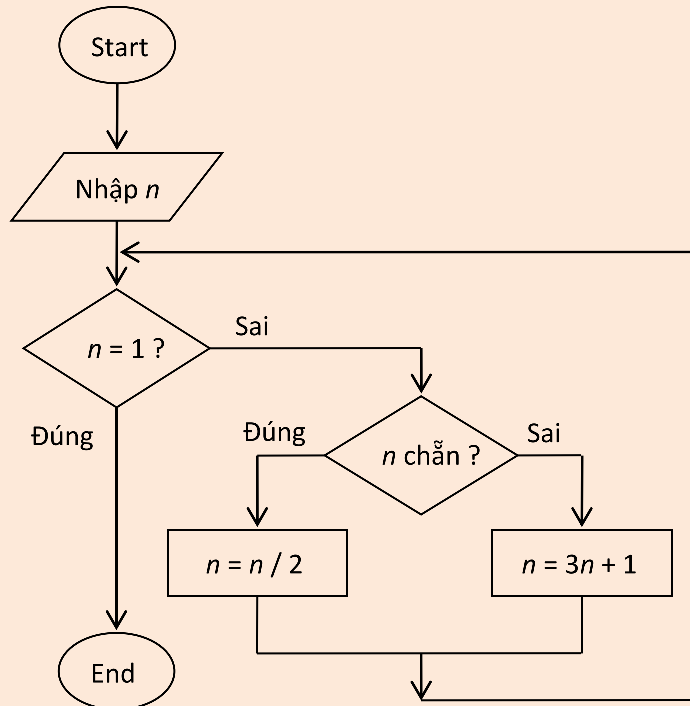
Ta đã thấy rằng số lần lặp của các lệnh lặp có thể hữu hạn hoặc vô hạn. Nếu số lần lặp là hữu hạn thì "sớm hay muộn" vòng lặp sẽ dừng còn nếu số lần lặp là vô hạn thì vòng lặp sẽ không dừng được mà ta gọi là lặp vô hạn. Lặp vô hạn thường là nguyên nhân phổ biến làm chương trình bị treo. Do đó ta luôn muốn kiểm tra để đảm bảo rằng các vòng lặp không bị lặp vô tận. Trong một số trường hợp việc kiểm tra này là khá dễ nhưng trong một số trường hợp khác thì rất khó, thậm chí là bất khả thi.

Ta xét một quá trình lặp rất đơn giản như sau: cho *n* là một số nguyên dương, lặp lại quá trình sau đây cho đến khi *n* là 1 thì dừng:

- Nếu *n* chẵn: *n* = *n* / 2.
- Nếu *n* lẻ: *n* = 3*n* + 1.

Quá trình lặp này có thể được mô tả rõ hơn bằng lưu đồ như sau:

Hình 2.5.7 – Lưu đồ quá trình lặp 3*n* + 1



Người ta đã thử với rất nhiều giá trị *n* (nguyên dương) khác nhau và đều thấy rằng quá trình lặp trên là dừng, chẳng hạn với *n* = 12 thì quá trình lặp sẽ cho *n* lần lượt là 12, 6, 3, 10, 5, 16, 8, 4, 2, 1 và dừng. Tuy nhiên không có ai chứng minh được là quá trình trên luôn dừng với mọi trường hợp của *n* (nguyên dương). Câu hỏi "Có trường hợp nào của *n* để vòng lặp trên lặp vô tận hay không?" được gọi là nghi vấn Collatz[7] hay nghi vấn 3*n* + 1 mà chưa có ai trả lời được.

Chương trình sau đây thực hiện quá trình lặp 3*n* + 1 với giá trị *n* do người dùng nhập. Bạn hãy chạy thử, biết đâu giải được nghi vấn này (nghĩa là tìm được giá trị *n* làm chương trình lặp vô tận):) Bạn cũng nghiên cứu mã để thành thạo cách viết các vòng lặp trong C nhé.

Mã 2.5.1 – Chương trình thực thi quá trình lặp 3n + 1

```
#include <stdio.h>

int main()
{
```

[7] Bạn có thể đọc qua bài viết "Collatz conjecture" trên trang Wikipedia (https://end.wikipedia.org/wiki/Collatz_conjecture).

```
5       int n;
6       printf("Nhap n: ");
7       scanf("%d", &n);
8       while(n > 0)
9       {
10          int num = 0;
11          printf("%d ", n);
12          while(n != 1)
13          {
14              if(n % 2 == 0)
15                  n = n / 2;
16              else
17                  n = 3*n + 1;
18              num++;
19              printf("%d ", n);
20          }
21          printf("\nSo lan lap: %d\n", num);
22          printf("\nNhap n: ");
23          scanf("%d", &n);
24      }
25  
26      return 0;
27  }
```

Mà vòng while ngoài cùng (Dòng 8) có lặp vô tận không vậy?

BÀI TẬP

Bt 2.5.1 Vẽ lưu đồ và viết chương trình tính các tổng sau với *n* là số nguyên dương được cho.

a) $S(n) = 1 + 1/2 + 1/3 + ... + 1/n$.
b) $S(n) = 1 + 1/2! + 1/3! + ... + 1/n!$.

Bt 2.5.2 Viết chương trình tính các tổng sau với *x* là số thực và *n* là số nguyên không âm được cho.

a) $S(x, n) = 1 + x + x^2 + ... + x^n$.
b) $S(x, n) = 1 + x + x^2/2! + ... + x^n/n!$.

Bt 2.5.3 Vẽ lưu đồ và viết chương trình kiểm tra một số nguyên dương được cho có là số nguyên tố hay không. (Một số nguyên ≥ 2 được gọi là số nguyên tố khi nó chỉ có 2 ước số là 1 và chính nó).

Bt 2.5.4 Viết chương trình xuất ra tất cả các số nguyên tố không quá số nguyên dương *n* được cho.

Bt 2.5.5 Viết chương trình phân tích một số nguyên ≥ 2 ra tích các thừa số nguyên tố.

BÀI 2.6

Bài này là bài tạp kĩ. Ta sẽ tìm hiểu một vài vấn đề "không ăn nhập" nhưng quan trọng của C tới giờ. Trước hết là toán tử độc nhất vô nhị của C: *toán tử điều kiện*. Đây là toán tử ba ngôi duy nhất trong C. Toán tử này gồm hai kí hiệu (? và `:`) được viết theo cú pháp như sau:

<*E*1> ? <*E*2> : <*E*3>

với *E*1, *E*2, *E*3 là các biểu thức (khác `void`) bất kì. Toán tử này là toán tử ba ngôi vì nó nhận 3 toán hạng (3 biểu thức con là *E*1, *E*2, *E*3). Toàn bộ biểu thức trên gọi là *biểu thức điều kiện* và nó được lượng giá như sau:

Lượng giá *E*1:

- nếu được giá trị đúng (khác 0) thì lượng giá *E*2 và giá trị của *E*2 sẽ là giá trị của toàn bộ biểu thức mà không lượng giá *E*3.
- nếu được giá trị sai (là 0 hoặc 0.0) thì lượng giá *E*3 và giá trị của *E*3 sẽ là giá trị của toàn bộ biểu thức mà không lượng giá *E*2.

Ngữ nghĩa trên có thể nói nôm na là: "Nếu *E*1 đúng thì được *E*2 còn không thì được E3".

Biểu thức điều kiện thường giúp ta viết mã cô đọng hơn trong một số trường hợp mà không cần dùng lệnh `if-else`. Chẳng hạn lệnh sau giúp xuất ra trị tuyệt đối của x (với x là biến `double`):

```
printf("%lf", x >= 0 ? x : -x);
```

mà nếu không dùng nó thì bạn có thể phải viết dài dòng hơn bằng lệnh `if-else` như:

```
if(x >= 0)
    printf("%lf", x);
else
    printf("%lf", -x);
```

Hoặc là nếu ta cần bỏ giá trị lớn nhất của *a*, *b* vào *m* (nghĩa là *m* = max{*a*, *b*}) thì ta có thể viết gọn như sau[1]:

```
m = (a > b) ? a : b;
```

Cặp ngoặc tròn ở trên cũng có thể bỏ đi vì toán tử có điều kiện có độ ưu tiên rất thấp, chỉ hơn các toán tử gán. Tiện thể, nó có tính kết hợp phải[2].

[1] Bạn tự viết lại bằng cách dùng lệnh `if-else`.

[2] Thật ra toán tử này cũng còn điều kì bí mà ta sẽ biết sau.

Bạn đã biết về các toán tử luận lý ở Bài 2.1 như toán tử phủ định (!), hoặc (||), và (&&) với ý nghĩa cho ở Bảng 2.1.1. Tuy nhiên một điều quan trọng nữa cần biết là toán tử || và && được C lượng giá theo cách gọi là *lượng giá tắt*. Biểu thức *A* && *B* được lượng tắt như sau:

Lượng giá *A*:

- nếu được giá trị đúng thì lượng giá *B* và giá trị của *B* sẽ là giá trị của toàn bộ biểu thức (Đúng && *B* sẽ có chân trị như *B*).
- nếu được giá trị sai thì giá trị của toàn bộ biểu thức là sai (mà không lượng giá *B*) (Sai && *B* sẽ có chân trị là Sai mà không phụ thuộc chân trị của *B*).

Tương tự, biểu thức *A* || *B* được lượng giá tắt như sau:

Lượng giá *A*:

- nếu được giá trị đúng thì giá trị của toàn bộ biểu thức là Đúng (mà không lượng giá B) (Đúng || *B* sẽ có chân trị là Đúng mà không phụ thuộc chân trị của *B*).
- nếu được giá trị sai thì lượng giá *B* và giá trị của *B* sẽ là giá trị của toàn bộ biểu thức (Sai || *B* sẽ có chân trị như *B*).

Nói theo kiểu biểu thức điều kiện thì *A* && *B* tương đương với *A* `?` *B* `: 0` còn *A* || *B* tương đương với *A* `? 1 :` *B*.

Ta thấy cách lượng giá trên có chân trị hoàn toàn giống với cách lượng giá "đầy đủ" (nghĩa là, lượng giá *A*, lượng giá *B* và theo bảng chân trị sẽ có chân trị cho *A* && *B*, *A* || *B*), hơn nữa cách lượng giá tắt sẽ nhanh hơn. Trường hợp *A*, *B* là các biểu thức không có hiệu ứng lề thì cách lượng giá tắt và cách lượng giá đủ không có khác biệt (ngoại trừ lượng giá tắt sẽ có thể nhanh hơn). Chẳng hạn biểu thức: `1 <= i && i < 10` (để kiểm tra điều kiện $1 \le i < 10$) được lượng giá tắt hay đủ đều như nhau hơn nữa biểu thức đó cũng tương đương với `i < 10 && 1 <= i` (*A* && *B* có giống *B* && *A*). Hoặc biểu thức ngược lại: `i < 1 || i >= 10` (để kiểm tra điều kiện ngược lại: $I \notin [1, 10)$) được lượng giá tắt hay đủ đều như nhau hơn nữa biểu thức đó cũng tương đương với `i >= 10 || i < 1` (*A* || *B* cũng có chân trị giống *B* || *A*).

Tuy nhiên trường hợp *A*, *B* có hiệu ứng lề hoặc *B* phụ thuộc *A* thì 2 cách lượng giá không còn giống nhau nữa. Cụ thể với cách lượng giá tắt thì *A* luôn được lượng giá nhưng *B* có thể được hoặc không được lượng giá tùy theo giá trị của *A*. Có thể nói rằng với cách lượng giá tắt thì ngữ nghĩa của *A* && *B* là "có *A* và hơn nữa còn có *B*" tức là *B* được giả định rằng *A* đúng còn của *A* || *B* là "có *A* còn không thì có *B*" tức là *B* được giả định rằng *A* sai. Với cách lượng giá đầy đủ thì ngữ nghĩa của *A* && *B* là "có *A* và có *B*" còn của *A* || *B* là "có *A* hoặc có *B*" tức là *A* và *B* không liên quan gì với nhau.

Chẳng hạn trong C ta thường thấy cách vận dụng lượng giá tắt biểu thức luận lý như:

```
if(a != 0 && b/a > 1)
    b = a;
```

Nhờ cách lượng giá tắt mà biểu thức `b/a > 1` chỉ được lượng giá khi $a \neq 0$ do đó tránh được lỗi chia cho 0. Nếu lượng giá theo cách đủ thì chân trị vẫn như vậy nhưng trường hợp $a = 0$ (chân trị là Sai) thì cách lượng giá đủ sẽ gặp lỗi chia cho 0. Với cách lượng giá đủ ta phải viết dài hơn như sau để tránh lỗi chia cho 0:

```
if(a != 0)
    if(b/a > 1)
        b = a;
```

Lưu ý nếu viết biểu thức luận lý thành `b/a > 1 && a != 0` thì ta vẫn bị lỗi chia cho 0 khi $a = 0$. Như vậy đây là cách viết có chủ đích, hơn nữa cách viết này khá phổ biến trong C nên ta phải lưu ý (để hiểu người khác viết và để viết cho người khác hiểu).

C có một toán tử cũng khá kì lạ nữa là toán tử phẩy (`,`) với cú pháp như sau:

<*E*1>, <*E*2>

Biểu thức trên được C lượng giá như sau: đầu tiên lượng giá biểu thức *E*1, sau đó lượng giá biểu thức *E*2 và giá trị của *E*2 là giá trị của toàn bộ biểu thức. Như vậy *E*1 được lượng giá nhưng giá trị của nó không được dùng, cho nên thường thì *E*1 có hiệu ứng lề. Toán tử phẩy là toán tử có độ ưu tiên thấp nhất trong C (sau cả các toán tử gán) và có tính kết hợp trái. Các biểu thức *E*1, *E*2 là các biểu thức bất kì nên nó có thể là biểu thức phẩy khác dẫn đến ta có dãy biểu thức phẩy. Chẳng hạn biểu thức:

```
i = 1, j = 2, k = i + j
```

sẽ được lượng giá lần lượt từ trái qua do đó `i` sẽ có giá trị 1, `j` sẽ có giá trị 2 và `k` sẽ có giá trị 3 (1 + 2) và giá trị của toàn bộ biểu thức trên là l-value `k` mà nếu dùng như r-value sẽ có giá trị là 3 (giá trị của `k` sau khi gán).

Toán tử phẩy cho phép ta "nối" nhiều biểu thức lại để được một biểu thức cũng kiểu kiểu như lệnh ghép cho phép ta "nối" nhiều lệnh lại thành một lệnh. Biểu thức phẩy thật ra ít khi được dùng trừ vài trường hợp như cách dùng sau đây trong vòng lặp `for`:

```
int i, j;
for(i = 1, j = 10; i < j; i++, j--)
    printf("%s(%d + %d)", (i == 1 ? "" : " + "), i, j);
```

Tương truyền thì Gauss đã nghĩ ra cách tính tổng các số từ 1 đến 100 bằng cách gom cặp số như sau:

$1 + 2 + \ldots + 99 + 100$
$= (1 + 100) + (2 + 99) + \ldots + (50 + 51) = 101 \times 50 = 5050$

Đoạn mã trên mô tả cách gom cặp số của tổng từ 1 đến 10[3]. Nếu bạn không hiểu cách Gauss đã làm thì cũng chẳng sao:) Quan trọng là ta đã dùng toán tử phẩy trong tiền lệnh và hậu lệnh của vòng lặp `for` những nơi yêu cầu một biểu thức.

Có một lệnh rẽ nhánh nữa trong C là lệnh `switch` (bên cạnh `if`-khuyết và `if`-đủ) mà tôi sẽ giới thiệu ở đây bằng bài toán "Tìm số ngày trong tháng". Ta biết rằng các tháng (dương lịch) có 31 ngày là: tháng 1, 3, 5, 7, 8, 10, 12; riêng tháng 2 có 28 ngày (hoặc 29 ngày nếu năm nhuận) còn lại là các tháng có 30 ngày. Giả sử `thang` (tháng) là biến nguyên cho biết tháng mấy trong năm và chứa giá trị phù hợp (1 ≤ `thang` ≤ 12). Ta có thể dùng `if` tầng để làm như sau.

Mã 2.6.1 – Đoạn mã tìm số ngày trong tháng bằng `if` tầng

```
if(thang == 1 || thang == 3 || thang == 5
        || thang == 7 || thang == 8
        || thang == 10 || thang == 12)
    printf("31 ngay");
else if(thang == 2)
    printf("28 ngay"); // gia su khong phai nam nhuan
else
    printf("30 ngay");
```

Bằng cách dùng lệnh switch ta có thể viết lại đoạn mã trên như sau.

Mã 2.6.2 – Đoạn mã tìm số ngày trong tháng bằng lệnh `switch`

```
switch(thang)
{
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        printf("31 ngay");
        break;
    case 2:
        printf("28 ngay");
        break;
    default:
        printf("30 ngay");
        break;
}
```

Cú pháp chung của lệnh `switch` là:

`switch`(<Biểu thức số nguyên *E*>)
{
`case` <Hằng số 1>: <Các lệnh 1>
`case` <Hằng số 2>: <Các lệnh 2>

[3] Bạn có thể sửa lại thành 1 đến 100 nhưng xuất ra hơi dài.

```
    ...
    case <Hằng số N>: <Các lệnh N>
    default: <Các lệnh mặc định>
}
```

với switch, case, default là các từ khóa. *E* là biểu thức số nguyên (chỉ biểu thức có giá trị là số nguyên mới được chấp nhận, kể cả biểu thức có giá trị là số thực cũng không được chấp nhận) và được gọi là biểu thức của switch. Trong thân của switch, mỗi trường hợp (case) được xác định bằng một <Hằng số> và <Các lệnh> tương ứng. <Hằng số> là một hằng số nguyên (như 1, 3, … ở mã trên) hoặc một *biểu thức hằng số nguyên*, tức là biểu thức chỉ có các số hạng là hằng số nguyên như 10 + 2*5. Lưu ý, các hằng kí tự như 'A', 'a', … cũng là các hằng số nguyên vì chúng mô tả mã ASCII của các kí tự tương ứng. <Các lệnh> là một dãy lệnh (có thể không có như trường hợp case 1: hay case 3:, … ở mã trên). *Trường hợp mặc định* (default:) là trường hợp đặc biệt có thể có cũng có thể không. Tất cả các trường hợp phải khác nhau (giá trị của các <Hằng số> phải khác nhau, default nếu có thì chỉ có 1) và có thể liệt kê theo thứ tự tùy ý.

Ngữ nghĩa của switch là: lượng giá biểu thức của switch (biểu thức *E*) để được giá trị nguyên sau đó so giá trị này với các hằng (<Hằng số>) trong các trường hợp (case), nếu có một hằng trùng với giá trị này thì thực thi <Các lệnh> tương ứng của trường hợp đó, nếu không có hằng trùng hợp thì thực thi <Các lệnh> của trường hợp mặc định (default) nếu có. Lưu ý là <Các lệnh> là dãy lệnh bất kì (có thể rỗng) và nếu không có lệnh nhảy nào trong <Các lệnh> thì khi được thực hiện xong (mà nếu rỗng thì không làm gì) sẽ tiếp tục thực hiện <Các lệnh> của trường hợp bên dưới (kể cả default). Chẳng hạn <Các lệnh> của case 10: là rỗng nên nếu thang có giá trị là 10 thì case 10: được thực thi (không có <Các lệnh> nên không làm gì) xong sẽ thực thi <Các lệnh> của case bên dưới là case 12:. Tương tự, nếu thang là 1, thì nó sẽ chuyển xuống case dưới là case 3:, rồi chuyển xuống case dưới là case 5:, …, rồi case 12:. Thường thì trong <Các lệnh> của các trường hợp có dùng *lệnh nhảy* break. Lệnh này cú pháp cực kì đơn giản:

```
break;
```

với break là từ khóa. Lệnh này dùng trong switch sẽ giúp kết thúc thực thi switch (một cách hình ảnh là nhảy ra khỏi switch). Chẳng hạn ở case 12: trên (mà như vậy là cho cả các case 1:, case 3:, …, case 10:) thì <Các lệnh> bao gồm 2 lệnh: đầu tiên là lệnh đơn printf("31 ngay"); và sau đó là lệnh nhảy break;. Lệnh nhảy này giúp kết thúc switch sau khi ta đã xuất ra 31 ngay. Nếu không có lệnh nhảy này thì sau khi thực thi printf("31 ngay"); sẽ tiếp tục thực thi <Các lệnh> của case 2:, tức là tiếp tục xuất ra 28 ngay. Tương tự trong case 2: và default: ta cũng có lệnh nhảy break sau khi xuất ra số ngày tương ứng. Thật ra thì break trong default: ở trên

là không cần thiết vì default: ở trên được liệt kê cuối cùng nên khi thực thi <Các lệnh> của nó thì cũng kết thúc switch. Tuy nhiên vì ta có thể viết default: trước các case (tùy ý thứ tự) nên trong trường hợp đó ta cũng phải có break.

Bạn có nắm được ngữ nghĩa của switch không đó? Nếu không nắm được thì … càng tốt:) Tôi thực sự thấy rằng nó quá lằng nhằng. Nói chung, dùng if (với các dạng khác nhau như if lồng, if tầng, …) thì tốt hơn. Thật ra có tình huống mà dùng switch sẽ thuận tiện hơn và tôi sẽ chỉ cho bạn thấy khi phù hợp[4], còn từ đây tới đó thì lời khuyên là: bạn không nên dùng switch. Lệnh switch còn có nhược điểm là nó chỉ rẽ nhánh với các giá trị cụ thể của biểu thức số nguyên. C không cho phép mô tả chẳng hạn một khoảng các số trong các trường hợp. Nói chung là chỉ có thể so sánh bằng với hằng số nguyên khi dùng switch. Những điều này thì lại không bị hạn chế trong if (so sánh >, >=, … với các biểu thức bất kì, và kết hợp các điều kiện với toán tử luận lý). Chẳng hạn bạn đã thấy cách ta phân loại chỉ số BMI bằng cách dùng if tầng trong Bài 2.4 (Mã của Bảng 2.4.2). Ta không thể viết lại bằng cách dùng switch được.

Cũng lưu ý các case rỗng (tức là không có <Các lệnh>) thì thường được viết lên cùng một dòng cho dễ nhìn (như cách viết trên), nhưng hiển nhiên là ta có thể viết ra trên các dòng khác nhau.

Ta đã thấy cách dùng lệnh nhảy break để nhảy ra khỏi switch. Ta cũng có thể dùng lệnh này để nhảy ra khỏi các lệnh lặp (while, do-while, for). Bên cạnh đó một lệnh nhảy nữa cũng hay dùng trong các vòng lặp là lệnh continue[5] cho phép ta "bỏ qua phần còn lại của thân lệnh lặp". Ngữ nghĩa của break và continue trong các vòng lặp được minh họa qua các mã C và lưu đồ trong Hình 2.6.1.

Ở mã trong Hình 2.6.1 thì biểu thức điều kiện của while là 1 mà khi dùng như biểu thức luận lý thì luôn có giá trị Đúng. Hiển nhiên vòng lặp while như vậy sẽ không bao giờ thoát (nếu không có lệnh nhảy). Ở đây tôi dùng nó để sơ đồ đơn giản vì ta chỉ tập trung vào break và continue mà thôi. Từ lưu đồ ta thấy lệnh break cho phép ta nhảy khỏi vòng lặp (kết thúc vòng lặp) còn lệnh continue cho phép ta bỏ qua phần còn lại của thân lặp (và tiếp tục với lặp lần tiếp theo). Có thể nói rằng continue nhảy đến cuối thân vòng lặp (còn break nhảy đến ngay sau vòng lặp, tức là lệnh kế lệnh lặp). break và continue cho ta khả năng *điều khiển luồng thực thi* của các vòng lặp linh hoạt, đa dạng hơn. Chẳng hạn break cho phép ta có thể có nhiều điểm thoát khỏi vòng lặp hay thoát khỏi ở giữa mà không phải là đầu hay cuối như trong while và do-while. Với lệnh nhảy linh hoạt hơn, goto,

---

[4] Và khi đó bạn mới có thể hiểu switch rõ ràng hơn và sẽ không còn thấy rối nữa, nhất là khi bạn đã biết về trùm của lệnh nhảy goto.

[5] continue không được dùng trong lệnh switch.

ta có thể nhảy đi bất kì đâu tùy thích[6] còn lệnh nhảy `return` thì cho phép ta "nhảy" khỏi hàm. Ta sẽ tìm hiểu rõ hơn các ông trùm này sau.

Hình 2.6.1 – Minh họa ngữ nghĩa của `break` và `continue`

Mã C | **Lưu đồ**

```
while(1)
{
    <S1>
    if(<E>)
        break;
    <S2>
}
```

S1
E ?
Đúng
Sai
S2

```
while(1)
{
    <S1>
    if(<E>)
        continue;
    <S2>
}
```

S1
E ?
Đúng
Sai
S2

Đoạn mã sau đây minh họa cách mà `break` hay được dùng trong vòng lặp. `i` và `n` là các biến `int`. Đoạn mã kiểm tra xem một số nguyên n ≥ 2 là số nguyên tố hay hợp số. Một số nguyên ≥ 2 được gọi là số nguyên tố nếu nó không chia hết cho số nào ngoài 1 và chính nó. Ngược lại thì nó được gọi là hợp số. Như vậy chỉ cần n chia hết cho một số nào đó trong tập {2, 3, …, n –

[6] Nói hơi quá.

1} thì nó sẽ là hợp số. Vòng lặp `for` giúp ta tìm số `i` như vậy. Vì ta chỉ cần thấy một `i` như vậy là đủ kết luận `n` là hợp số nên ta sẽ dùng lệnh nhảy `break` để thoát khỏi vòng lặp (mà không cần kiểm tiếp nữa). Lệnh xuất sau `for` có sử dụng biểu thức điều kiện nhưng không quan trọng (bạn có thể viết dài hơn bằng `if-else`). Quan trọng là nếu `break` không được thực thi trong `for` thì vòng lặp kết thúc bởi điều kiện `i < n` sai mà đó chỉ có thể là lúc `i` = `n` (vì ta đã giả sử `n` ≥ 2). Mà khi `break` không được thực thi nghĩa là `n` không chia hết cho số nào trong tập {2, ..., `n` - 1}. Như vậy sau khi kết thúc vòng lặp thì `i` = `n` khi và chỉ khi `n` là số nguyên tố. Tương tự khi `break` được thực thi thì `i` < `n` và khi đó `n` là hợp số. Ta cũng thấy rằng vòng lặp `for` chắc chắn kết thúc (với giả định `n` ≥ 2).

Mã 2.6.3 – Đoạn mã minh họa dùng break trong vòng lặp

```
for(i = 2; i < n; i++)
{
    if(n % i == 0)
        break;
}
printf("%d la %s\n", n, i < n ? "hop so"
    : "so nguyen to");
```

Để kết thúc bài tạp kĩ này, ta sẽ bàn thêm một chủ đề cực kì quan trọng: *kĩ năng debug chương trình*. Như bạn đã thấy trong Bài 1.6, một chương trình chạy được không có nghĩa là không có lỗi. Nó chỉ không có lỗi biên dịch nhưng vẫn có thể có bug. Để có thể phát hiện và chỉnh sửa chương trình để loại bỏ bug ta cần phải hiểu tường tận "hành vi" của chương trình, nghĩa là theo dõi được từng bước thực thi và thay đổi trạng thái của chương trình qua tương tác với người dùng. Rất may là các IDE đều hỗ trợ cộng cụ (gọi là *debugger*) giúp lập trình viên thực hiện công việc theo vết này. Kĩ năng debug (mà một phần trong đó là kĩ năng sử dụng debugger) thực sự là một kĩ năng thực hành quan trọng mà mọi lập trình viên cần phải thuần thục. Phụ lục A.7 hướng dẫn cơ bản cách debug chương trình.

MỞ RỘNG 2.6 – Vòng lặp tương tác người dùng

Chắc hẳn ở quê ai cũng đã một lần chơi "Bầu Cua" mỗi khi Tết đến. Ngoại trừ chuyện biến tướng thành cờ bạc thì đây có lẽ là một nét văn hóa Tết của người Việt giống như việc đốt pháo, lì xì, ... Trò chơi có bàn cờ gồm 6 ô ứng với 6 "linh vật" thân quen của người Việt là Tôm, Cua, Bầu, Cá, Gà và Nai. Sau khi người chơi "đặt" (cược) tiền vào các ô này, nhà cái sẽ rung 3 viên xúc xắc có 6 mặt ứng với 6 linh vật trên. Nếu trong 3 viên xúc xắc "có" (xuất hiện) linh vật mà người chơi đã cược, họ sẽ lấy lại tiền cược và nhà cái phải "chung" (trả) số tiền bằng số lần xuất hiện linh vật đó nhân

với số tiền cược. Nếu linh vật người chơi đặt không xuất hiện thì nhà cái sẽ “ăn” (lấy) số tiền cược cho linh vật đó. Chẳng hạn nếu nhà cái rung ra 3 Bầu (cả 3 xúc xắc đều ra mặt Bầu) thì nhà cái sẽ ăn tất cả số tiền trong 5 linh vật còn lại và người chơi đặt Bầu sẽ nhận lại số tiền đã đặt cho Bầu cùng với số tiền chung của cái là 3 lần số tiền đã đặt cho Bầu. Bạn đọc thêm bài viết “Lắc bầu cua” trên trang Wikipedia Việt để biết thêm[7]. Sau đây là chương trình C mô phỏng trò chơi này.

Mã 2.6.4 – Chương trình chơi “Bầu Cua”

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define DUNG    0
#define TOM     1
#define CUA     2
#define BAU     3
#define CA      4
#define GA      5
#define NAI     6
#define RUNG    7

void xuat(int tom, int cua, int bau, int ca, int ga, int nai) {
    if(tom + cua + bau + ca + ga + nai == 0)
        printf("Ban chua dat cuoc");
    else {
        printf("Ban dang dat cuoc: ");
        if(tom > 0)
            printf("Tom (%d$) ", tom);
        if(cua > 0)
            printf("Cua (%d$) ", cua);
        if(bau > 0)
            printf("Bau (%d$) ", bau);
        if(ca > 0)
            printf("Ca (%d$) ", ca);
        if(ga > 0)
            printf("Ga (%d$) ", ga);
        if(nai > 0)
            printf("Nai (%d$) ", nai);
    }
    printf("\nBan muon (0-Dung, 1-Tom, 2-Cua, 3-Bau, 4-Ca, 5-Ga, 6-Nai, 7-Rung): ");
}
```

[7] https://vi.wikipedia.org/wiki/ Lắc_bầu_cua.

```
int main() {
    int cmd;
    int tom, cua, bau, ca, ga, nai;
    int sotom, socua, sobau, soca, soga, sonai;
    int cuoc;

    tom = cua = bau = ca = ga = nai = 0;
    srand(time(0));
    while(1) {
        xuat(tom, cua, bau, ca, ga, nai);
        scanf("%d", &cmd);
        switch(cmd) {
        case TOM:
            printf("Ban dat Tom bao nhieu tien? ");
            scanf("%d", &tom);
            break;
        case CUA:
            printf("Ban dat Cua bao nhieu tien? ");
            scanf("%d", &cua);
            break;
        case BAU:
            printf("Ban dat Bau bao nhieu tien? ");
            scanf("%d", &bau);
            break;
        case CA:
            printf("Ban dat Ca bao nhieu tien? ");
            scanf("%d", &ca);
            break;
        case GA:
            printf("Ban dat Ga bao nhieu tien? ");
            scanf("%d", &ga);
            break;
        case NAI:
            printf("Ban dat Nai bao nhieu tien? ");
            scanf("%d", &nai);
            break;
        case RUNG:
            cuoc = tom + cua + bau + ca + ga + nai;
            sotom = socua = sobau = soca = soga = sonai = 0;
            printf("Ban da dat cuoc %d$\n", cuoc);
            printf("Ket qua rung: ");
            for(int i = 1; i <= 3; i++) {
                int t = rand() % 6 + TOM;
                switch(t) {
            case TOM: sotom++; printf("Tom "); break;
```

```
80              case CUA: socua++; printf("Cua "); break;
81              case BAU: sobau++; printf("Bau "); break;
82              case CA: soca++; printf("Ca "); break;
83              case GA: soga++; printf("Ga "); break;
84              case NAI: sonai++; printf("Nai "); break;
85                  }
86              }
87              cuoc += sotom > 0 ? sotom*tom : -tom;
88              cuoc += socua > 0 ? socua*cua : -cua;
89              cuoc += sobau > 0 ? sobau*bau : -bau;
90              cuoc += soca > 0 ? soca*ca : -ca;
91              cuoc += soga > 0 ? soga*ga : -ga;
92              cuoc += sonai > 0 ? sonai*nai : -nai;
93              printf("\nBan con %d$\n", cuoc);
94              tom = cua = bau = ca = ga = nai = 0;
95              break;
96          case DUNG:
97              printf("Chuc mung nam moi!");
98              return 0;
99          }
100     }
101
102     return 0;
103 }
```

Điều ta học trong phần này không phải là chơi "Bầu Cua" mà là vòng lặp `while` ở Dòng 43 (đến Dòng 100). Đây là một khuôn mẫu của *vòng lặp tương tác người dùng*. Chương trình sẽ lặp đi lặp lại việc nhận lệnh[8] từ người dùng và thực thi công việc tương ứng cho đến khi người dùng muốn dừng (hay đóng) chương trình. Biểu thức điều kiện của vòng lặp `while` là hằng số 1. Nó luôn có giá trị là 1 mà khi dùng như giá trị luận lý thì là đúng. Do đó vòng lặp này sẽ lặp vô tận. Tuy nhiên bên trong vòng lặp ta sẽ dùng lệnh đặc biệt để kết thúc chương trình khi người dùng muốn.

Chương trình bắt đầu mỗi phiên tương tác (tức là mỗi lần lặp) bằng việc hiển thị trạng thái/thông tin hiện hành của chương trình mà ở đây là trạng thái đặt cược các ô của người chơi. Vì việc hiển thị này khá dài dòng nên ta đã tách riêng nó ra trong hàm `xuat` và đưa ra lời gọi đến nó ở Dòng 44. Các biến `tom`, ..., `nai` (được khai báo ở Dòng 37 và khởi gán giá trị 0 ở Dòng 41) chứa số tiền mà người chơi đặt cược ở ô Tôm, ..., Nai. Hàm `xuat` (Dòng 14 đến Dòng 33) xuất ra thông tin số tiền mà người chơi đang cược ở từng ô (hoặc thông báo chưa cược gì, Dòng 16, nếu không có

[8] Thuật ngữ lệnh (command) ở đây được hiểu là yêu cầu của người dùng. Phân biệt với lệnh (statement) của chương trình C.

ô nào được cược, nghĩa là số tiền là 0). Hàm này, ở Dòng 32, cũng xuất ra “menu lệnh” tức là thông báo cho biết “lệnh” nào tương ứng với “phím ấn” nào. Như thông báo cho biết, người dùng có thể đưa ra 8 “lệnh” (tức là 8 yêu cầu) khác nhau với “phím nhấn” tương ứng từ 0 đến 7. Ở đây để “nhấn phím lệnh” thì người dùng nhập số nguyên tương ứng (từ 0 đến 7). Biến `cmd` sẽ chứa “mã lệnh” đó nhờ lệnh nhập ở Dòng 45.

Sau khi nhận mã lệnh, chương trình sẽ thực thi công việc tương ứng. Để làm điều này ta có thể dùng cấu trúc `if`-tầng nếu có ít mã lệnh hoặc dùng `switch` nếu có nhiều mã lệnh. Ở đây ta sẽ dùng `switch` như là một minh họa của cách mà `switch` hay được dùng[9]. Các `case` tương ứng với các mã lệnh với thân là công việc tương ứng với lệnh đó. Ở đây để mã rõ ràng, gợi nhớ, ta đã dùng các hằng tượng trưng `TOM`, `CUA`, … cho các mã lệnh tương ứng thay vì dùng con số khó nhớ. Các hằng tượng trưng này được `#define` từ Dòng 5 đến Dòng 12. Hiển nhiên là hằng số mà bạn dùng để `#define` cho chúng phải tương ứng với mã lệnh. Cũng lưu ý là `TOM`, …, `NAI` phải tương ứng với các số (nguyên) liên tiếp theo đúng thứ tự Tôm, …, Nai vì một lý do mà ta sẽ thấy sau.

Lệnh đặc biệt nhất là lệnh đóng (hay dừng) chương trình. Trong trường hợp đó (Dòng 96) ta sẽ kết thúc hàm `main` bằng lệnh `return` (Dòng 98) và do đó kết thúc chương trình[10]. Các lệnh khác tương ứng với việc người dùng chọn đặt cược cho ô nào đó, chẳng hạn đặt ô Tôm ở Dòng 47. Lệnh đặt biệt khác là lệnh “Rung” ở Dòng 71. Biến `cuoc` chứa tổng số tiền người dùng cược ở các ô. Các biến `sotom`, …, `sonai` được khởi gán 0 ở Dòng 73 sẽ chứa số lần mặt Tôm, …, Nai xuất hiện ở 3 xúc xắc sau khi rung. Ta “mô phỏng” việc rung 3 xúc xắc (vòng `for` ở Dòng 76) bằng cách dùng số ngẫu nhiên. Để làm việc này, đầu tiên ta phải “khởi động” “bộ tạo số ngẫu nhiên” bằng cách dùng hàm `srand` (trong thư viện `stdlib.h`), đối số của hàm này là giá trị trả về của hàm `time` (trong thư viện `time.h`) với đối số `0`[11] như Dòng 42 cho thấy. Sau đó ta dùng hàm `rand` để phát sinh một số nguyên không âm ngẫu nhiên và đưa về phạm vi từ 0 đến 5 (ứng với 6 ô) nhờ phép chia dư cho 6 và cho tương ứng với một ô trong các ô Tôm, …, Nai bằng cách cộng thêm cho `TOM` như Dòng 77 cho thấy. Lưu ý rằng tương ứng các số 0, …, 5 với các ô Tôm, …, Nai chỉ có được khi các hằng tượng trưng `TOM`, …, `NAI` được `#define` là các số nguyên liên tiếp bắt đầu với `TOM`. Bạn tự nghiên cứu đoạn mã tiếp sau đó của lệnh “Rung” nhé.

---

[9] Thậm chí trong trường hợp này, tôi vẫn thích dùng `if`-tầng hơn.

[10] Trước đó ta cũng không quên an ủi người dùng khi mà họ đã thua sạch tiền bằng lệnh xuất ở Dòng 97:)

[11] Ta sẽ tìm hiểu nhiều hơn các hàm này sau.

Một lựa chọn khác để "mã" các lệnh là dùng kí tự thay vì số nguyên. Khi đó biến `cmd` sẽ có kiểu là `char`, lệnh nhập ở Dòng 45 thay bằng nhập kí tự (chẳng hạn dùng `scanf` với `%c`) và các hằng của `case` thay bằng hằng kí tự tương ứng. Trong các chương trình dạng đồ họa[12] thì cách tương tác của người dùng sẽ phong phú và thuận tiện hơn như là dùng chuột, menu lệnh, nút lệnh trên thanh công cụ, phím tắt, … Khi đó vòng lặp thực thi còn được gọi là *vòng lặp thông điệp*.

BÀI TẬP

Bt 2.6.1 Cho biến nguyên `d` hãy dùng biểu thức điều kiện để in ra dấu của `d` là âm, không, hay dương tùy theo giá trị của `d`.

Bt 2.6.2 Viết chương trình cho nhập tháng và xuất tên tiếng Anh của tháng bằng `if` tầng và `switch`. Tên tiếng Anh của 12 tháng lần lượt là: January, February, March, April, May, June, July, August, September, October, November và December.

Bt 2.6.3 Giả sử `i`, `j`, `k` là các biến `int` lần lượt chứa các giá trị 1, 5, 10. Hãy cho biết kết quả xuất ra của các đoạn mã sau:

a)
```
printf("%d ", i < j || ++j < k);
printf("%d %d %d", i, j, k);
```
b)
```
printf("%d ", i-- && ++j < k);
printf("%d %d %d", i, j, k);
```
c)
```
printf("%d ", (i = j) && (j = k));
printf("%d %d %d", i, j, k);
```
d)
```
printf("%d ", --i || ++j && ++k);
printf("%d %d %d", i, j, k);
```
e)
```
printf("%d ", --i && ++j || ++k);
printf("%d %d %d", i, j, k);
```

Bt 2.6.4 Viết mã C cho lưu đồ ở Hình 2.3.6 bằng cách dùng lệnh `break`.

Bt 2.6.5 Viết lại Mã 2.6.4 trong phần Mở rộng để không còn dùng lệnh `switch`.

Bt 2.6.6 Bổ sung vào Mã 2.6.4 để khi người dùng chọn "Dừng" thì chương trình đưa ra câu hỏi "Bạn có muốn dừng không? (Nhấn *y*/*Y* để dừng)" và nếu người dùng nhấn kí tự *y* (hoặc *Y*) thì chương trình sẽ dừng còn không thì chương trình sẽ tiếp tục.

[12] Là các chương trình ta hay thấy với hình ảnh, màu sắc, chuột, menu, toolbar, …

Bt 2.6.7 Tương tự Mã 2.6.4 trong phần Mở rộng hãy viết chương trình mô phỏng trò chời Xì dách (tham khảo bài viết "Xì dách" trên trang Wikipedia Việt, https://vi.wikipedia.org/wiki/Xì_dách)[13].

[13] Nhắc lại lần nữa, tài liệu này không khuyến khích cờ bạc.

BÀI 2.7

Ta đã viết chương trình "Hello người dùng" ở Mã 1.2.1 của Bài 1.2. Trong bài này ta cũng viết một chương trình như vậy nhưng là phiên bản tiếng Việt, chương trình "Chào người dùng". Chương trình của ta cũng cho người dùng nhập tên và đưa ra lời chào đến họ. Khác biệt ở đây là người dùng được phép nhập tên tiếng Việt và chương trình xuất ra lời chào tiếng Việt đến họ. Chẳng hạn, kết xuất sau đây là cho người dùng Vũ Quốc Hoàng[1].



Toàn bộ mã của chương trình được cung cấp kèm tài liệu này. Bạn chạy thử, nghiên cứu theo hướng dẫn trong Bài này để hiểu rõ nhé. Sau đây là hàm `main` của chương trình (rất giống Mã 1.2.1).

Mã 2.7.1 – Hàm `main` của chương trình "Chào người dùng"

```
int main()
{
    char ten[100];

    printf("Ban ten gi? ");
```

[1] It's me:)

```
    gets(ten);

    xuatChuoi("CHA`O");
    xuatChuoi(ten);

    return 0;
}
```

Để chương trình có thể làm việc với tiếng Việt thì ta phải giải quyết ít nhất 3 vấn đề sau đây:

(1) Cách thức để người dùng nhập chuỗi tiếng Việt.
(2) Cách thức biểu diễn (và lưu trữ/trao đổi) chuỗi tiếng Việt.
(3) Cách thức hiển thị chuỗi tiếng Việt.

Có nhiều cách từ đơn giản đến phức tạp, từ tự phát đến chuẩn hóa mà ta sẽ bàn sau. Ở đây ta sẽ chọn cách đơn giản và khá tự phát.

Thứ nhất, để *nhập chuỗi tiếng Việt*, người dùng sẽ nhập chuỗi theo dạng *VIQR*[2]. Đây là dạng mô tả tiếng Việt khá phổ biến trước đây[3] và vẫn còn sử dụng đến giờ. Theo đó để mô tả các chữ cái tiếng Việt có dấu ta dùng qui tắc sau:

Bảng 2.7.1 – Bảng qui tắc mô tả dấu chữ cái tiếng Việt theo VIQR

Dấu	*Ký hiệu*	*Ví dụ*
Trăng	(	A(→ Ă
Mũ	^	A^ → Â
Râu	+	O+ → Ơ
Sắc	'	A' → Á
Huyền	`	A` → À
Hỏi	?	A? → Ả
Ngã	~	A~ → Ã
Nặng	.	A. → Ạ
Đ/đ	DD/dd	DD → Đ

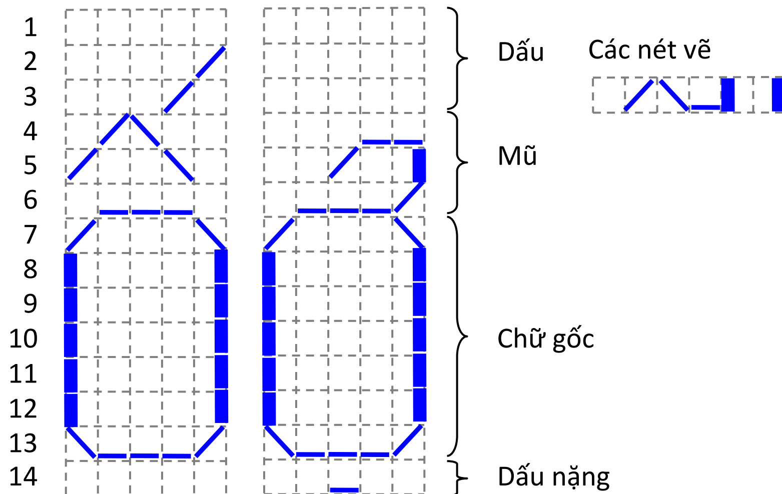
Chẳng hạn để mô tả chuỗi tiếng Việt "Vũ Quốc Hoàng" ta dùng chuỗi VIQR là "Vu~ Quo^'c Hoa`ng" hay để mô tả chuỗi "Việt Nam đất nước mến yêu" ta dùng chuỗi VIQR là "Vie^.t Nam dda^'t nu+o+'c me^'n ye^u". Lưu ý là dấu mũ (trăng, mũ, râu) phải được ghi trước dấu thanh (sắc, huyền, hỏi, ngã, nặng) nếu cả hai cùng xuất hiện trên một chữ cái.

[2] Bạn hãy tìm hiểu rõ hơn định dạng này tại bài viết VIQR trên trang Wikipedia (https://vi.wikipedia.org/wiki/VIQR).
[3] Khi mã Unicode chưa được hỗ trợ rộng rãi.

Thứ hai, ta sẽ dùng chính chuỗi VIQR để *biểu diễn chuỗi tiếng Việt*[4]. Như vậy ở chương trình trên, khi được hỏi tên, người dùng sẽ nhập tên theo qui tắc VIQR, chuỗi VIQR này sau đó được chứa trong biến chuỗi `ten`.

Thứ ba, ta phải tìm cách *hiển thị được các chữ cái tiếng Việt có dấu*. Trong bài này ta sẽ "thiết kế" một "*font*"[5] chữ tiếng Việt để hỗ trợ việc hiển thị các chữ cái tiếng Việt (có dấu hay không dấu). Để đơn giản ta chỉ thiết kế cho chữ in hoa và *kích thước chữ là cố định*[6]. Mỗi chữ cái sẽ có kích thước 5 × 14, rộng 5 và cao 14 ô kí tự. Nghĩa là ta sẽ dùng một khối gồm 5 ô theo chiều ngang và 14 ô theo chiều dọc, mà ta gọi là lưới chữ, để vẽ nên một chữ cái. Mỗi ô trong lưới sẽ dùng một trong 6 "*nét vẽ*": nét trống (không có gì), nét nghiêng trái, nét nghiêng phải, nét dưới, nét đậm trái và nét đậm phải. 14 dòng trong lưới được đánh số từ trên xuống với số thứ tự được gọi là số hiệu của dòng. Để cho đơn giản ta cũng phân lưới ra 4 phần cố định cho một chữ cái gồm: phần dấu (các dòng từ 1 đến 4), phần mũ (các dòng từ 4 đến 6), phần chữ gốc (các dòng từ 6 đến 13) và phần cho dấu nặng (dòng 14). Hình 2.7.1 minh họa lưới chữ với các phần và các nét vẽ cho chữ cái Ố và Ợ.

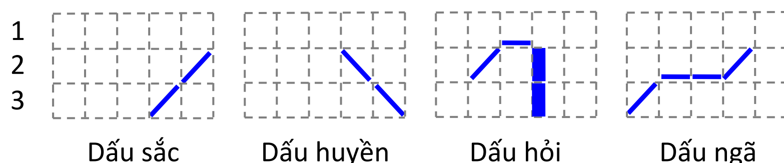
Hình 2.7.1 – Lưới chữ của các chữ cái Ố và Ợ



Các chữ cái (nguyên âm) tiếng Việt có thể không có dấu thanh hoặc có một trong 5 dấu thanh là sắc, huyền, hỏi, ngã và nặng. Trừ dấu nặng được bố trí riêng ở dòng 14 trong lưới (như minh họa ở Hình 2.7.1), 4 dấu còn lại được bố trí ở phần dấu (dòng 1 đến 3) trong lưới như Hình 2.7.2.

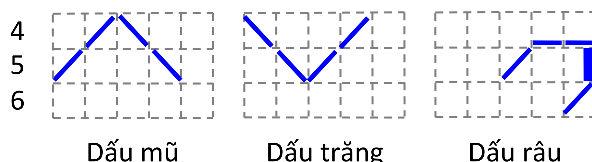
[4] Một lựa chọn tốt hơn là dùng mã UTF-8 mà ta sẽ bàn sau.
[5] Thuật ngữ font chữ dùng để chỉ kiểu dáng, kích thước của chữ.
[6] Thuật ngữ là monospaced hay fixed-width font.

Hình 2.7.2 – Nét vẽ của các dấu thanh sắc, huyền, hỏi và ngã



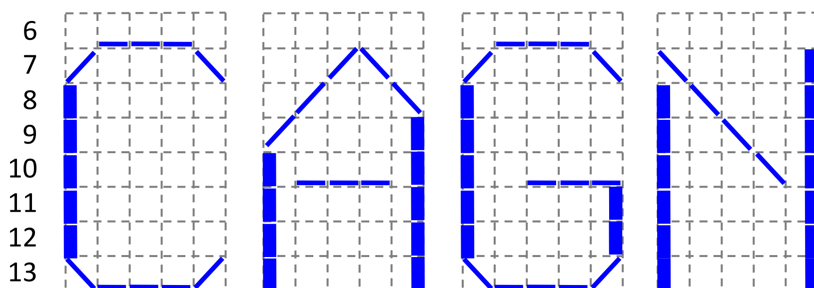
Các dấu mũ: mũ (như trong Â), trăng (như trong Ă) và râu (như trong Ơ) được bố trí ở phần mũ (dòng 4 đến 6) trong lưới như Hình 2.7.3. Lưu ý là chỉ có dấu râu mới dùng dòng 6 của lưới.

Hình 2.7.3 – Nét vẽ của các dấu mũ



Phần chữ gốc được bố trí ở dòng 6 đến dòng 13 và phải được thiết kế cho đầy đủ bộ chữ cái (hoa) A, B, C, ..., Z. Tuy nhiên hiện chương trình mới thiết kế mẫu cho vài chữ cái như minh họa ở Hình 2.7.4. Lưu ý là các chữ cái O, Q, C, G, ... có dùng dòng 6 của lưới chữ, còn các chữ cái khác như A, H, M, N, ... không dùng dòng 6.

Hình 2.7.4 – Chữ gốc của các chữ cái C, A, G và N



Về mặt kĩ thuật thì 5 × 14 ô của lưới chữ được xuất ra bằng 5 × 14 kí tự tương ứng với nét vẽ của từng ô: ô trống ứng với kí tự space, nét nghiêng trái ứng với kí tự xổ trái /, nét nghiêng phải ứng với kí tự xổ phải \, nét dưới ứng với kí tự gạch dưới _, nét đậm trái ứng với kí tự mã 0xdd và nét đậm phải ứng với kí tự mã 0xde. Cũng nhớ rằng các kí tự space, /, _ là các kí tự bình thường trong C nhưng kí tự \ là kí tự đặc biệt ta cần dùng kí pháp `'\\'` hay kí tự mã 0xdd dùng kí pháp `'xdd'` và kí tự mã 0xde dùng kí pháp `'xde'`.

Chẳng hạn phần chữ gốc của chữ cái A (dòng 7 đến dòng 13) được xuất ra bởi đoạn mã sau.

Mã 2.7.2 – Đoạn mã xuất phần chữ gốc của chữ cái A

```
if(c == 'A')
{
    switch(dong)
    {
        case 7: printf("  /\\ "); break;
        case 8: printf(" /  \\"); break;
        case 9: printf("/   \xde"); break;
        case 10: printf("\xdd___\xde"); break;
        case 11: printf("\xdd   \xde"); break;
        case 12: printf("\xdd   \xde"); break;
        case 13: printf("\xdd   \xde"); break;
    }
}
```

Bạn đối chiếu với bản thiết kế nét vẽ của chữ cái A ở Hình 2.7.4 để hiểu rõ nhé. Ở đây c chứa chữ cái (kí tự) gốc và dong cho biết số hiệu dòng của lưới chữ. Thật sự thì Mã 2.7.2 là phần thi công cho thiết kế chữ A ở Hình 2.7.4.

Hàm xuatChuoi giúp xuất một dòng chữ tiếng Việt để trong tham số chuoi viết theo qui tắc VIQR. Về mặt logic, sẽ tự nhiên hơn khi ta xuất ra từng chữ cái trong chuỗi, với mỗi chữ cái ta xuất ra từng dòng (từ dòng 1 đến dòng 14) của lưới chữ, nghĩa là ta dùng *vòng lặp lồng* với vòng lặp bên ngoài là theo từng chữ cái trong chuỗi còn vòng lặp bên trong là theo từng dòng trong lưới chữ. Tuy nhiên do hạn chế về kĩ thuật, việc xuất ra màn hình yêu cầu phải theo từng dòng: xong dòng trên mới đến dòng dưới và không thể quay lên lại dòng trên[7]. Do đó ta sẽ làm hơi gượng gạo là xuất ra từng dòng trong lưới chữ, với mỗi dòng thì xuất ra tương ứng các nét vẽ của dòng đó cho từng chữ cái, nghĩa là vòng lặp bên ngoài là theo từng dòng trong lưới chữ còn vòng lặp bên trong là theo từng chữ cái trong chuỗi. Khuôn mẫu của hàm xuatChuoi như vậy là:

Mã 2.7.3 – Khuôn mẫu xuất dòng chữ tiếng Việt trong hàm xuatChuoi

```
void xuatChuoi(const char chuoi[])
{
    for(int dong = 1; dong <= 14; dong++)
    {
        int i = 0;
        while(chuoi[i] != '\0' && chuoi[i] != '\n')
        {
            // phân tích chữ cái tiếng Việt
            //   từ vị trí i trong chuoi theo VIQR
```

[7] Ta sẽ học cách vượt qua hạn chế này sau.

```
10              // và xuất chữ cái tiếng Việt
11              //   với chữ gốc, dấu mũ và thanh tương ứng
12              // và tăng i phù hợp
13
14              printf(" ");    // khoang cach chu: 1 SPACE
15          }
16          printf("\n");
17      }
18  }
```

Dòng 1 cho thấy cách khai báo tham số là chuỗi. Đặc biệt, từ khóa `const` nói rằng tham số chuỗi đó (`chuoi`) không được thay đổi trong hàm. Điều này tương tự như việc đặt từ khóa `const` trước một khai báo biến để cấm việc thay đổi giá trị của biến sau khi khởi tạo (và như vậy nó trở thành hằng `const`) mà bạn đã thấy trong Bài 1.4 (Mã 1.4.1). Thật sự, thân của hàm `xuatChuoi` chỉ đọc các kí tự trong `chuoi` mà không thay đổi nó. Bạn cũng xem lại Bài 1.7 để biết về chuỗi.

Một chữ cái tiếng Việt theo VIQR sẽ thuộc 1 trong 4 trường hợp sau:

1) Chữ không dấu
2) Chữ chỉ có dấu thanh
3) Chữ chỉ có dấu mũ
4) Chữ vừa có dấu mũ vừa có dấu thanh

Trong cả 4 trường hợp thì chữ gốc nằm ở vị trí `i`. Trường hợp (2) thì dấu thanh nằm ở vị trí `i+1`, trường hợp (3) và (4) thì dấu mũ nằm ở vị trí `i+1`, trường hợp (4) thì dấu thanh nằm ở vị trí `i+2` (nhớ là dấu mũ phải ở trước dấu thanh trong trường hợp (4)). Trong trường hợp (1) thì chữ cái tiếp theo ở vị trí `i+1` nên `i` tăng 1 đơn vị, trường hợp (2) và (3) thì chữ cái tiếp theo ở vị trí `i+2` nên `i` tăng 2 đơn vị còn trường hợp (4) thì chữ cái tiếp theo ở vị trí `i+3` nên `i` tăng 3 đơn vị. Để xuất một chữ cái với gốc chữ, dấu mũ và dấu thanh tương ứng ta dùng hàm `xuatChu`. Sau khi xuất chữ thì Dòng 14 xuất khoảng cách giữa 1 chữ mà ở đây là một kí tự Space[8]. Sau khi xuất xong tất cả các kí tự của dòng chữ thì Dòng 16 giúp xuống dòng[9]. Hai hàm `laDauMu` và `laDauThanh` giúp kiểm tra có phải là kí tự mô tả dấu mũ hay dấu thanh theo qui tắc VIQR không (xem lại Bảng 2.7.1). Cũng lưu ý, để tránh mã cứng các kí hiệu này ta dùng các hằng tượng trưng như sau:

```
#define TRANG   '('
#define MU      '^'
#define RAU     '+'
#define SAC     '\''
```

[8] Nếu muốn tăng khoảng cách giữa các chữ thì bạn sửa chỗ này để xuất nhiều kí tự Space hơn.

[9] Cũng vậy, nếu muốn tăng khoảng cách giữa các dòng chữ thì bạn sửa chỗ này để xuất nhiều kí tự xuống dòng (`'\n'`) hơn.

```
#define HUYEN   '`'
#define HOI     '?'
#define NGA     '~'
#define NANG    '.'
```

Hàm `xuatChu` giúp xuất các nét vẽ cho dòng có số hiệu trong tham số `dong` của lưới chữ cho chữ cái tiếng Việt với chữ gốc, dấu mũ và dấu thanh lần lượt trong tham số `c`, `mu` và `thanh`. Dấu mũ hay dấu thanh có thể không có mà khi đó tham số tương ứng, `mu` hay `thanh`, sẽ nhận giá trị là 0. Khuôn mẫu của hàm `xuatChu` như sau.

Mã 2.7.4 – Khuôn mẫu xuất chữ cái tiếng Việt trong hàm xuatChu

```
1 void xuatChu(char c, char mu, char thanh, int dong)
2 {
3     c = toupper(c);
4 
5     // Xét số hiệu dòng (dong)
6     // Với từng phần (dấu, mũ, chữ gốc, dấu nặng)
7     //   xuất các nét của thanh, mũ, chữ gốc tương ứng
8 }
```

Ta đã dùng các *lệnh rẽ nhánh lồng* nhau nhiều kiểu để xét các trường hợp trên. Đặc biệt, phần gốc chữ sẽ rất dài do ta phải xét tất cả các trường hợp chữ cái A, B, C, …, Z khác nhau (với từng chữ là từng dòng từ 7 đến 13 trong lưới chữ). Lưu ý đến việc xuất các nét vẽ của dòng 6 trong lưới chữ vì phần mũ của dấu râu và phần gốc chữ của một số chữ cái (O, Q, C, G, …) có dùng dòng này. Như đã nói, để đơn giản, "font" chữ của ta chỉ hỗ trợ chữ cái in hoa nên ta dùng hàm `toupper` (Dòng 3) để chuyển chữ gốc thành chữ hoa trước khi xử lý tiếp. Hàm `toupper` nằm trong thư viện chuẩn `ctype.h`.

Thật sự thì mã của chương trình này quá dông dài. Bạn sẽ biết cách viết nó cô đọng (tự nhiên và đẹp) hơn nhiều khi bạn học thêm các kĩ thuật trong các bài sau. Hiện giờ đó là dịp để bạn rèn luyện kĩ năng dùng các cấu trúc lặp và rẽ nhánh (lồng nhau). Cũng nhớ là mã của chương trình chưa đầy đủ, bạn cần bổ sung phần xuất gốc chữ cho các chữ cái khác nữa. Hơn nữa bạn thử tự mình sáng tạo một "font" chữ khác. Bài này thật sự là một cơ hội để bạn thử kiểm tra năng khiếu … hội họa của mình:)

BÀI TẬP

Bt 2.7.1 Chương trình "Chào người dùng" còn chưa đầy đủ phần chữ gốc (và dòng 6 của lưới chữ) và chữ cái Đ. Hãy bổ sung cho hoàn chỉnh.

Bt 2.7.2 Hãy sáng tạo một "font" chữ khác cho chương trình "Chào người dùng". Một lựa chọn đơn giản là font chữ với chỉ một trong hai nét là trống và đầy ở mỗi ô của lưới chữ[10]. Font chữ như vậy thường thấy trong các bộ

[10] Tương ứng là kí tự Space và kí tự có mã 0xdb của Code page 437.

hiển thị ma trận điểm. (Tham khảo bài viết “Dot-matrix display” trên trang Wikipedia[11].)

Bt 2.7.3 Tạo “font chữ 7-segment” để hiển thị các kí số như bộ hiển thị LED 7-đoạn. (Tham khảo bài viết “Seven-segment display”[12].)

[11] https://en.wikipedia.org/wiki/Dot-matrix_display
[12] https://en.wikipedia.org/wiki/Seven-segment_display

BÀI 3.1

Ta đã biết cách viết chương trình C tính toán (cộng, trừ, nhân, chia, …) trên các số nguyên và thực. Còn tính toán trên các số hữu tỉ thì sao? *Số hữu tỉ* là số có dạng *a*/*b*, trong đó *a*, *b* là các số nguyên với *b* ≠ 0, *a* được gọi là *tử số* và *b* được gọi là *mẫu số*. Số hữu tỉ được gọi một cách bình dân là *phân số*. Chẳng hạn, phân số 1/3 có tử là 1 và mẫu là 3. Số hữu tỉ (trên máy) có lợi thế hơn số thực (trên máy) là cho kết quả tính toán chính xác (trừ trường hợp tràn số). Chẳng hạn phân số 1/2 chia cho phân số 3/2 được phân số 1/3, còn số thực 0.5 chia cho số thực 3.5 được số thực gần đúng là 0.333333.

Tuy nhiên, phân số không phải là *kiểu dữ liệu được hỗ trợ sẵn* trong C: không có từ khóa khai báo kiểu nào như `int` (cho số nguyên) hay `double` (cho số thực), không có toán tử nào hay hàm nào trong thư viện chuẩn tính toán trên phân số, cũng không có hỗ trợ nào trong nhập/xuất cho phân số như `%d` (cho số nguyên) hay `%lf` (cho số thực). Vậy ta phải làm sao? Đây là tình huống hay gặp: không phải cái gì cũng được C hỗ trợ sẵn. Ta phải tự mình làm lấy, nhưng không phải tự mình làm tất cả[1]. Vì tử và mẫu đều là các số nguyên nên ta tận dụng kiểu nguyên đã được C hỗ trợ sẵn:

- Dùng hai biến nguyên `int` cho một phân số.
- Dùng các phép toán trên số nguyên để tính toán trên phân số. Chẳng hạn, ta đều quen thuộc với công thức: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$.
- Dùng các hỗ trợ *nhập xuất có định dạng* của C. Và đây là chủ đề của bài này.

Bạn hãy thử chương trình sau[2].

Mã 3.1.1 – Chương trình tính toán trên phân số

```
#include <stdio.h>
int main()
{
    int a, b, c, d;
    char op;

    printf("Nhap bieu thuc phan so: ");
    scanf("%d / %d %c %d / %d", &a, &b, &op, &c, &d);

```

[1] Nhớ chiến lược "tận dụng tối đa những gì đã có".

[2] Từ Tầng này thì bạn không chỉ được phép mà nên thử nghiệm, sáng tạo. Chương trình nêu ra chỉ là mẫu, bạn có thể thêm, bớt và chỉnh sửa.

```
10      if(op == '+')
11          printf("Ket qua: %d/%d\n", a*d + c*b, b*d);
12  
13      return 0;
14  }
```

Chẳng hạn ta tính tổng của phân số 1/3 với 2/3 như sau.

```
Nhap bieu thuc phan so: 1/3 + 2/3↵
Ket qua: 9/9
```

Đây dĩ nhiên là kết quả đúng nhưng kết quả tốt hơn nên là 3/3 hay thậm chí là 1. Chương trình trên cũng chỉ mới tính tổng, bạn hãy mở rộng để tính hiệu, tích và thương của hai phân số.

Ở trên, ta đã dùng 4 biến nguyên cho 2 phân số (a, b là tử và mẫu của phân số thứ nhất; c, d là tử và mẫu của phân số thứ hai). Các toán tử +, -, *, / là các kí tự nên ta dùng biến kiểu char để chứa (biến op ở trên). Dòng 10 cho thấy cách kiểm tra giá trị trong biến op (là toán tử mà người dùng đã nhập) có phải là một kí tự cụ thể nào đó không ('+' chẳng hạn).

Điểm quan tâm của ta là lệnh nhập ở Dòng 8. Đối số đầu tiên của scanf được gọi là *chuỗi định dạng*. Nó xác định dạng thức (khuôn mẫu) của chuỗi mà người dùng cần nhập (gọi gọn là *chuỗi nhập*). Trong chuỗi định dạng, các kí hiệu như %d, %c[3] được gọi là *đặc tả chuyển đổi*, các kí tự khác (ngoại trừ khoảng trắng) được gọi là kí tự thông thường. Các đặc tả chuyển đổi yêu cầu phần chuỗi nhập tương ứng có dạng nào đó (%d là một số nguyên viết ở cơ số 10, %c là một kí tự) và chuyển đổi phần chuỗi nhập đó thành dạng dữ liệu tương ứng rồi bỏ vào biến có kiểu phù hợp, được cung cấp sau đó, theo thứ tự. Chẳng hạn, lời gọi scanf trên có chuỗi định dạng là "%d / %d %c %d / %d", xác định rằng chuỗi nhập có dạng: chuỗi số nguyên[4] (mà sẽ được chuyển đổi thành số nguyên và bỏ vào biến nguyên a, là biến đầu tiên sau chuỗi nhập), rồi đến các khoảng trắng (nếu có), rồi đến kí tự /, rồi đến các khoảng trắng (nếu có), rồi đến chuỗi số nguyên (mà sẽ được chuyển đổi thành số nguyên và bỏ vào biến nguyên b, là biến kế tiếp), rồi đến các khoảng trắng (nếu có), rồi đến kí tự (mà mã ASCII của kí tự đó sẽ được đưa vào biến op, mà ta dùng nó để xác định toán tử), rồi đến các khoảng trắng (nếu có), ...

Quá trình so khớp và chuyển đổi trên được gọi là *đối sánh*. scanf xuất phát từ đầu, sẽ đi qua lần lượt các kí tự trong chuỗi định dạng và đối sánh với phần tương ứng trong chuỗi nhập của người dùng cho đến khi không so khớp được hoặc đến cuối chuỗi định dạng. Chi tiết như sau:

---

[3] Viết sát kí tự d, c với kí tự %.

[4] Bạn phải phân biệt chuỗi số nguyên -12, là một chuỗi gồm 3 kí tự liên tiếp -, 1, 2, với số nguyên -12 (âm 12).

(1) Nếu là kí tự thông thường thì sẽ so khớp kí tự đó với kí tự tương ứng trong chuỗi nhập. Nghĩa là: nếu giống thì đọc qua kí tự đó trong chuỗi nhập và tiếp tục đối sánh qua kí tự kế tiếp còn nếu không giống thì không so khớp được và dừng quá trình đối sánh.
(2) Nếu là *kí tự khoảng trắng* (space, tab, xuống dòng) thì sẽ đọc qua dãy kí tự khoảng trắng liên tiếp (không, một hoặc nhiều) trong chuỗi nhập.
(3) Nếu là đặc tả chuyển đổi thì tùy theo đặc tả đó mà đọc một hoặc nhiều kí tự liên tiếp trong chuỗi nhập và chuyển đổi chuỗi đọc được thành dạng dữ liệu tương ứng (và bỏ vào biến tương ứng) hoặc dừng quá trình đối sánh khi đụng kí tự không phù hợp trong chuỗi nhập:
   - `%d`, `%i` xác định số nguyên viết theo cơ số 10: bỏ qua các kí tự trắng ở đầu cho đến khi đụng kí tự `+`, `-` hoặc kí số rồi đọc tiếp các kí số kế đó cho đến khi đụng kí tự không là kí số trong chuỗi nhập. Chuyển đổi chuỗi thành số nguyên kiểu `int`.
   - `%lf`, `%le`, `%lg` xác định số thực viết theo kiểu thông thường (dùng dấu chấm thập phân) hoặc viết theo *kiểu khoa học* (dạng mũ cơ số 10 chẳng hạn `2e-3`, là số thực 2×10^{-3}, tức là 0.003). Chuyển đổi chuỗi thành số thực kiểu `double`.
   - `%f`, `%e`, `%g` tương tự như (`%lf`, `%le`, `%lg`) nhưng chuyển đổi chuỗi thành số thực kiểu `float` (là kiểu số thực có độ chính xác kém hơn kiểu `double`).
   - `%c` đọc kí tự bất kì, kể cả kí tự trắng, chuyển thành mã ASCII tương ứng.
   - `%s` xác định một chuỗi: bỏ qua các kí tự trắng ở đầu cho đến khi đụng kí tự không phải kí tự trắng, rồi đọc tiếp các tự không phải là khoảng trắng.

Bạn cũng lưu ý là chuỗi nhập được dùng chung, xuyên suốt toàn bộ chương trình[5]. Khi chương trình bắt đầu chạy thì chuỗi nhập là rỗng. Khi chương trình gọi các hàm nhập (`scanf`, `gets`, …) thì nếu chuỗi nhập rỗng, chương trình sẽ dừng, đợi người dùng nhập chuỗi từ bàn phím, sau khi người dùng nhập chuỗi và nhấn phím Enter thì chuỗi đó (cùng với kí tự xuống dòng, `'\n'`, mã ASCII 10) sẽ được đưa vào chuỗi nhập và hàm nhập tiếp tục chạy trên chuỗi nhập đó. Ví dụ, với đoạn mã:

```
scanf("%d", &a);
scanf("%d", &b);
scanf("%c", &c);
```

[5] Nó được gọi là *input buffer*.

Chương trình đợi bạn nhập: nếu bạn nhập `10↵` thì a sẽ chứa giá trị 10; chương trình tiếp tục đợi bạn nhập: nhập `20↵` thì b sẽ chứa giá trị 20 và c chứa 10 (là mã ASCII của kí tự xuống dòng Enter). Nếu ngay từ đầu bạn nhập: `10 20↵` thì a, b, c sẽ chứa giá trị tương ứng như trên mà chương trình không dừng để đợi nhập như trên.

Một điều nữa: hàm `scanf` trả về giá trị là số lượng mục chuyển đổi được. Vì quá trình đối sánh có thể dừng khi xảy ra không khớp nên con số này có thể nhỏ hơn số đặc tả chuyển đổi (cũng là số lượng biến sau chuỗi định dạng trong lời gọi `scanf`). Chẳng hạn lời gọi `scanf` ở chương trình trên:

```
scanf("%d / %d %c %d / %d", &a, &b, &op, &c, &d);
```

nên được kiểm tra giá trị trả về, nếu là 5 thì tốt, còn ít hơn thì chuỗi nhập không đúng định dạng yêu cầu.

Cũng lưu ý là phần chuỗi nhập còn lại vẫn tiếp tục nằm trong chuỗi nhập mà những lệnh nhập sau đó có thể dùng mà không cần đợi người dùng nhập. Bạn có thể gọi `fflush(stdin)`[6] để xóa rỗng chuỗi nhập (và do đó chương trình sẽ đợi người dùng nhập khi gặp hàm nhập). Ví dụ với đoạn mã:

```
scanf("%d", &a);
gets(s);
```

nếu bạn nhập: `10hi↵` thì chuỗi nhập là `10hi\n` và `scanf` sẽ đọc `10` (a chứa giá trị 10), còn lại trong chuỗi nhập là `hi\n` và `gets` sẽ lấy chuỗi này mà không đợi người dùng nhập (biến chuỗi s sẽ chứa chuỗi `hi`). Bạn có thể sửa lại như sau để tránh tình huống đáng tiếc này:

```
scanf("%d", &a);
fflush(stdin);
gets(s);
```

Bạn hãy nghiền ngẫm các ví dụ sau để rõ hơn hoạt động của `scanf`. Giả sử a, b, c, d là các biến `int` và x, y là các biến `double` và `f` là biến `float` và ch là biến `char` và s là biến chuỗi (`char s[100]`).

Bảng 3.1.1 – Các ví dụ nhập có định dạng với `scanf`

Lệnh nhập	Chuỗi nhập	Kết quả và giải thích
`scanf("%d%x%lf%f", &a, &b, &x, &f);`	`45 2d 0.001 1e-3`	a = 45, b = 45, x = 0.001, f = 0.001 %d: cơ số 10, %x: cơ số 16 `%lf` cho `double`, `%f` cho float và nhập số thập phân hoặc số thực kiểu khoa học đều được

[6] Bạn sẽ rõ lời gọi bí ẩn này sau.

`scanf("%lf%f%lf", &x, &y, &f);`	`0.001 0.001 0.001`	x = 0.001, y, f chứa giá trị không có ý nghĩa `%lf` cho double, `%f` cho float
`scanf("%d/%d\n%d/%d", &a, &b, &c, &d);`	`1/ 2  3 /4`	a = 1, b = 2, c = 3, d giữ giá trị trước đó \n là kí tự trắng như space
`scanf("%d%x%o", &a, &b, &c);`	`45 2d 55`	a = 45, b = 45, c = 45 %d, %x, %o: số nguyên cơ số 10, 16, 8
`scanf("%i%i%i", &a, &b, &c);`	`55 0x55 055`	a = 55, b = 85, c = 45 `%i` cho số nguyên tổng quát: tiếp đầu ngữ `0x` cho số cơ số 16, tiếp đầu ngữ `0` cho số cơ số 6, 1-9 cơ số 10.
`scanf("%*d%c%*c%*c%s", &ch, s);`	`123abcd123 abc`	ch = 97 (`'a'`), s = "d123" %*... chuyển đổi nhưng không gán vào biến Ý nghĩa chuỗi định dạng trên: Đọc và bỏ qua số nguyên đầu tiên, kí tự tiếp theo bỏ vào c, bỏ qua 2 kí tự kế tiếp, còn lại (cho đến kí tự trắng) bỏ vào chuỗi s
`scanf("%*[abc]%s", s);`	`abacadeaba`	`s = "deaba"` `%[các kí tự]` : chuyển đổi chuỗi gồm một hoặc nhiều kí tự trong tập
`scanf("%*[^:]:%[^,]%*[^:=]%*c%d", s, &a);`	`Name : AB, a = 10`	s = "AB", a = 10 `%[^các kí tự]` : chuyển đổi chuỗi gồm một hoặc nhiều kí tự không nằm trong tập
`scanf("%d%d", &a);`	`10 20`	Thiếu biến chứa kết quả so với số đặc tả chuyển đổi Lỗi runtime
`scanf("%d", &a, &b);`	`10 20`	a = 10, b giữ giá trị cũ Lỗi logic, dư biến chứa kết quả

`scanf("%d %f", &x, &a);`	`10 10.5`	x, a chứa giá trị không có ý nghĩa Lỗi logic: sai kiểu biến so với đặc tả
`scanf("%d %d", &a, b);`	`10 20`	Lỗi runtime, cần đưa địa chỉ của b (tức là &b)

Ngược lại với nhập là xuất. Bạn đã thấy cách *xuất một chuỗi cố định* như Dòng 7, đó là một trường hợp đặc biệt của *xuất có định dạng* như Dòng 11 với hàm `printf`. Tham số đầu tiên cũng là chuỗi định dạng. Cũng tương tự như trong chuỗi định dạng của nhập: các kí tự thông thường sẽ được xuất ra màn hình còn nếu là đặc tả chuyển đổi thì xuất ra chuỗi tương ứng với giá trị được cho trong các biểu thức sau đó theo thứ tự. Các đặc tả chuyển đổi cũng như trong đặc tả chuyển đổi nhập. Nói chung việc xuất ra sẽ dễ dàng (và ít khả năng bị lỗi) hơn việc nhập vào do ta (tức là chương trình) chủ động. Cũng vậy bạn học hỏi qua các ví dụ sau để rõ hơn hoạt động của `printf`.

Bảng 3.1.2 – Các ví dụ xuất có định dạng với `printf`

Lệnh xuất	Kết quả xuất	Giải thích
`printf("%d,\n %f.\n%lf", 10 + 2, 10.5, 10.5);`	`12,` `  10.500000.` `10.500000`	10 + 2 là 12 Kí tự phẩy (,), chấm (.) và khoảng trắng (2 kí tự space) là các kí tự bình thường Mặc định thì số thực được xuất ra với 6 chữ số lẻ `%lf` cũng như `%f`
`printf("%i\n%x\n0x%X\n%o", 45, 45, 45, 45);`	`45` `2d` `0x2D` `55`	`%i` cũng như `%d` cơ số 10 `%x` (`%X`) xuất ra cơ số 16 `%o` xuất ra cơ số 8
`printf("%f\n%e\n%g\n%g", 10e-8, 10e-8, 10e-8, 10e-4);`	`0.000000` `1.000000e-007` `1e-007` `0.001`	`%f` in ra dạng thập phân `%e` in ra dạng khoa học `%g` chọn in ra dạng phù hợp
`printf("%4d\n%.1f", 12, 10.5);`	`  12` `10.5`	`%4d`: 4 xác định độ rộng tối thiểu, nếu không đủ kí tự sẽ có khoảng trắng đằng trước (canh phải) `%.1f`: in ra 1 chữ số lẻ thập phân
`printf("1\t2\nAB\x43`	`1       2`	`\n` xuống dòng, `\t` tab

`\t\x83\x84\x85");`	ABC âäà	\x?? in ra kí tự có mã ASCII tương ứng (cơ số 16)
`printf("\"...\x22\n\'...'\n1/2\n1\\2\n%%d");`	"..." '...' 1/2 1\2 %d	\" để xuất ra dấu " (mã ASCII là 22) Dấu ' là kí tự bình thường \\ cho kí tự \ %% cho kí tự %
`printf("%c %c\n%4s.\n%-4s.", 'H', "Hi", "Hi", "Hi");`	H P Hi. Hi .	%c kí tự (nhận chuỗi thì là kí tự đầu tiên) %s chuỗi %4s canh phải, %-4s canh trái.
`printf("%s", 'H');`		Lỗi runtime. %s cho chuỗi.
`printf("%d %d", 10);`	10 1719283104	Thiếu một biểu thức cho đặc tả chuyển đổi %d thứ hai. Một lỗi logic: con số thứ 2 xuất ra không có ý nghĩa (và có thể khác nhau tùy lúc chạy)
`printf("%d", 10, 20);`	10	Dư một biểu thức (2 biểu thức mà chỉ có một đặc tả). Lỗi logic: giá trị của biểu thức thứ 2 dư không được xuất ra
`printf("%d %f", 0.5, 20);`	0 0.000000	Không khớp biểu thức cần xuất với đặc tả (số thực với %d và số nguyên với %f). Lỗi logic: kết quả xuất ra không có ý nghĩa.
`int a = 2;` `printf("%d %d", a, &a);`	2 1245024	Lỗi logic hay gặp: có dấu & trước biến. Số thứ 2 xuất ra là địa chỉ của biến a (sẽ khác nhau tùy lúc chạy).

Tôi chắc là bạn không dễ chịu gì khi học bài này[7]. Quá chi tiết! Quá phiền phức! Quá rắc rối! Nhưng đây là một mặt trong bản chất của C (cũng như của việc lập trình nói chung). C là ngôn ngữ cấp thấp (so với các ngôn ngữ như Java, C#, Python, Prolog, …), có những thứ rất chi tiết mà bạn phải để ý. Điều này phản ánh bản chất gần gũi của C với ngôn ngữ máy (hay hợp ngữ).

[7] Thú thật là tôi cũng không hạnh phúc gì khi soạn bài này:)

Mục đích của bài này, suy cho cùng, là cho bạn thấy mặt phiền phức của C. Bạn sẽ phải chấp nhận đối mặt với nó (bạn còn gặp nó ở nhiều chủ đề khác của C) để lập trình với C. Thật ra chỉ cần bạn chịu nó (như bạn chịu trả giá khi đi chợ vậy[8]) thì cũng không quá khó. Hơn nữa bạn không cần phải nhớ tất cả những thứ đó đâu. Tôi sẽ chỉ cho bạn chỗ tra cứu nó khi cần. Bạn chỉ cần hiểu nguyên lý, chấp nhận nó và biết cách tra cứu là được. (Giống như bạn chỉ cần biết cách trả giá, chấp nhận trả giá nhưng không cần nhớ giá của tất cả các mặt hàng, bạn có cuốn sổ ghi giá bên cạnh, bạn chỉ cần biết cách tra sổ là được).

BÀI TẬP

Bt 3.1.1 Cho biết kết quả xuất ra của hai lệnh xuất sau và giải thích:

a) `printf("12345\r67\n");`
b) `printf("12345\b\b\b67\n");`

Bt 3.1.2 Giả sử a, b là các biến kiểu `int` và x, y là các biến kiểu `float` và các biến đều chứa giá trị 0. Cho biết giá trị của các biến sau khi thực hiện các lệnh nhập sau với từng chuỗi nhập được cho và phân tích:
Lệnh nhập:

a) `scanf("%f%d%d", &x, &a, &b);`
b) `scanf("%d%f%d", &a, &x, &b);`
c) `scanf("%d%d%f", &a, &b, &x);`
d) `scanf("%d%f%f", &a, &x, &y);`
e) `scanf("%f%d%f", &x, &a, &y);`
f) `scanf("%f%f%d", &x, &y, &a);`

Chuỗi nhập tương ứng:

a) `1.25 3 4`
b) `3 1.25 4`
c) `3 4 1.25`
d) `1.25 3.5 4`
e) `1.25 4 3.5`
f) `4 1.25 3.5`

Bt 3.1.3 Cho biết khác biệt của các chuỗi nhập sau là gì: `"%d/%d"`, `"%d /%d"`, `"%d/ %d"`, `"%d / %d"`.

Bt 3.1.4 Hoàn chỉnh chương trình tính toán trên phân số (Mã 3.1.1) bằng cách thêm các phép toán. Bạn có thể đơn giản các phân số khi tính toán không?

Bt 3.1.5 Hãy làm lại Bài tập 2.4.6 để nhập/xuất thời điểm theo định dạng HH:MM:SS. (Ví dụ: thời điểm bắt đầu ngày là 00:00:00 và thời điểm cuối cùng trong ngày là 23:59:59)

---

[8] Tôi thực sự tin rằng những người đi chợ giỏi là những người có khiếu lập trình C.

BÀI 3.2

Giả sử ta muốn xuất ra *n* dấu sao (dấu *), mỗi dấu sao trên một dòng thì ta có thể viết đoạn mã như sau.

Mã 3.2.1 – Đoạn mã xuất ra *n* dấu sao, mỗi dấu sao trên một dòng

```
for(int i = 1; i <= n; i++)
{
    printf("*");
    printf("\n");
}
```

Trong đó `n` là biến nguyên chứa số lượng dòng mong muốn. Dĩ nhiên là đoạn mã trên không chạy một mình được do thiếu hàm `main` và thiếu khai báo biến cũng như đặt giá trị cho `n`. Ta có thể ráp vào chương trình hoàn chỉnh chạy được như sau.

Mã 3.2.2 – Chương trình xuất ra *n* dấu sao, mỗi dấu sao trên một dòng

```
#include <stdio.h>
int main()
{
    int n = 5;

    for(int i = 1; i <= n; i++)
    {
        printf("*");
        printf("\n");
    }

    return 0;
}
```

Hiển nhiên là ta có thể cho người dùng nhập `n` thay vì đặt cứng một giá trị (như 5 ở trên[1]). Đoạn mã trên (Mã 3.2.1) quá đơn giản nhưng là khuôn mẫu của một lớp nhiều "bài toán". Giả sử ta muốn xuất ra một hình vuông các dấu sao có kích thước là *n*, nghĩa là *n* dòng mỗi dòng gồm *n* dấu sao, thì thế nào? Khuôn mẫu chung là:

Mã 3.2.3 – Đoạn mã mẫu xuất ra *n* dòng kí tự

```
for(int i = 1; i <= n; i++)
{
```

[1] Bạn có thể tăng giá trị của `n` lên để có hình hoành tráng hơn:)


```
    // Xuất các kí tự của dòng thứ i

    printf("\n");
}
```

Trong trường hợp đơn giản đầu tiên thì tất cả các dòng đều chỉ có một kí tự sao nên việc xuất các kí tự của dòng thứ `i` (với 1 ≤ `i` ≤ n) đơn giản là:

```
printf("*");
```

Với trường hợp của hình vuông thì mỗi dòng ta cần xuất ra n kí tự sao. Ta cũng có thể làm điều này dễ dàng bằng một vòng lặp `for` như sau:

```
for(int j = 1; j <= n; j++)
    printf("*");
```

Ráp đoạn mã này vào khuôn (tại đúng vị trí) ta có đoạn mã để xuất ra hình vuông như sau.

Mã 3.2.4 – Đoạn mã xuất ra hình vuông *n*×*n* dấu sao

```
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= n; j++)
        printf("*");

    printf("\n");
}
```

Để vẽ các hình khác nhau ta cần tính cách xuất các kí tự cho mỗi dòng tức ta cần viết mã cho công việc "Xuất các kí tự của dòng thứ `i`". Công việc này nói chung phụ thuộc vào giá trị của `i` (dòng nào) và n (số dòng, mà cả bài toán "Xuất hình" phụ thuộc). Ta nói rằng công việc được *tham số hóa* bởi *tham số* `i` và n. Điều đó có nghĩa là khi thực hiện công việc này ta sẽ dựa vào giá trị của `i` và n. Trường hợp đặc biệt thì nó không phụ thuộc vào giá trị của tham số, nghĩa là thực hiện giống nhau trong mọi trường hợp của tham số. Chẳng hạn trường hợp đầu tiên, công việc đó là xuất ra một dấu sao, không phụ thuộc giá trị của cả `i` lẫn n. Trường hợp thứ hai, công việc đó là xuất ra n dấu sao, không phụ thuộc giá trị của `i` nhưng phụ thuộc giá trị của n.

Ở Mã 3.2.4 ta thấy có một vòng lặp `for` bên trong thân một vòng lặp `for`, ta gọi là *vòng lặp lồng*. Nói chung khi khuôn mẫu là vòng `for` (hay các lệnh lặp khác) và công việc bên trong phụ thuộc vào biến lặp cũng được cài bằng vòng `for` (hay các lệnh lặp khác) thì ta có các vòng lặp lồng[2].

Trong đa số trường hợp, công việc sẽ phụ thuộc vào tất cả các tham số của công việc. Chẳng hạn để vẽ hình tam giác vuông cân như kết xuất dưới

[2] Lưu ý là ta cần đặt tên khác nhau cho các biến lặp trong các vòng lặp lồng.

đây ta thấy ở dòng thứ `i` (1 ≤ `i` ≤ n) ta cần xuất ra n – `i` kí tự trắng sau đó là `i` kí tự sao. Ta có thể rút ra qui tắc này từ nhận xét cho các dòng cụ thể: dòng 1 thì xuất ra n – 1 kí tự trắng rồi đến 1 kí tự sao, dòng 2 thì n – 2 kí tự trắng rồi đến 2 kí tự sao, ..., dòng cuối cùng (dòng n) thì không có kí tự trắng mà là n kí tự sao. Như vậy, việc xuất các kí tự của dòng thứ `i` trong trường hợp vẽ hình tam giác vuông cân có thể được viết như Mã 3.2.5.

```
    *
   **
  ***
 ****
*****
```

Mã 3.2.5 – Đoạn mã "Xuất các kí tự dòng thứ `i`" cho tam giác vuông cân

```c
for(int j = 1; j <= n - i; j++)
    printf(" ");
for(int k = 1; k <= i; k++)
    printf("*");
```

Bạn ráp vào khung chương trình trên và chạy thử nhé. Ở đây, công việc "Xuất các kí tự của dòng thứ `i`" phụ thuộc vào cả `i` lẫn n.

Phức tạp hơn nữa giả sử ta sẽ vẽ 3 hình tam giác "đều" chồng lên nhau như kết xuất dưới đây.

```
         *
        ***
       *****
      *******
     *********
    *         *
   ***       ***
  *****     *****
 *******   *******
********* *********
```

Ở đây ta sẽ xuất ra 2n dòng, n dòng đầu tiên cho tam giác ở trên và n dòng sau đó cho 2 tam giác dưới. Ta thấy qui luật cho n dòng đầu khá khác biệt với n dòng sau nên ta sẽ tách ra thành 2 lần xuất, lần 1 cho n dòng đầu và lần 2 cho n dòng sau (mà ta vẫn gọi là dòng 1, 2, ..., n cho dễ). Như vậy ta sẽ dùng khuôn mẫu trên 2 lần. Với việc tính toán "Xuất các kí tự của dòng thứ `i`" tương tự trường hợp tam giác cân, ta có đoạn mã xuất 3 tam giác đều như sau.

Mã 3.2.6 – Đoạn mã xuất 3 tam giác đều

```c
// n dòng đầu
for(int i = 1; i <= n; i++)
```

```
{
    for(int j = 1; j <= 2*n - i; j++)
        printf(" ");
    for(int j = 1; j <= 2*i - 1; j++)
        printf("*");
    printf("\n");
}
// n dòng sau
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= n - i; j++)
        printf(" ");
    for(int j = 1; j <= 2*i - 1; j++)
        printf("*");
    for(int j = 1; j <= 2*(n - i) + 1; j++)
        printf(" ");
    for(int j = 1; j <= 2*i - 1; j++)
        printf("*");
    printf("\n");
}
```

Bạn ráp vào hàm `main` với biến nguyên n được nhập giá trị và chạy thử nhé.

Ta qua một dạng khác: tính các *tổng hữu hạn*. Giả sử ta cần tính $S(x, n) = 1 + x + x^2 + ... + x^n$ với x là một số thực và n là số nguyên không âm. Ta viết $S(x, n)$ để ám chỉ S phụ thuộc x và n[3]. Cũng lưu ý là: $1 = x^0$ và $x = x^1$ nên S là tổng tất cả x_i với i chạy từ 0 đến n. Về kí hiệu Toán có thể viết $S = \sum_{i=0}^{n} x^i$ hay $S = \sum_{i=0}^{n} y(x,i)$ với $y(x,i) = x^i$. Các $y(x, i)$ được gọi là số hạng của tổng. Ta viết $y(x, i)$ vì giá trị này phụ thuộc vào x và i. Vận dụng vòng lặp `for` ta có khuôn mẫu chung của việc tính tổng hữu hạn $S = \sum_{i=0}^{n} y(x,i)$ như sau.

Mã 3.2.7 – Đoạn mã mẫu tính tổng hữu hạn

```
double S = 0;
for(int i = 0; i <= n; i++)
{
    double y;
    // Tính giá trị số hạng y(x, i) bỏ vào y
    S = S + y;
}
```

Công việc "Tính giá trị số hạng $y(x, i)$" được tham số theo x, và i. Ở đây $y(x, i) = x^i$ và có thể được cài đặt như sau:

```
    y = 1;
```

[3] Về mặt Toán thì S là hàm hai biến thực giá trị thực. Trong C ta cũng có thể viết S như là "hàm C" mà ta sẽ làm trong bài sau.

```
        for(int j = 0; j < i; j++)
            y = y * x;
```

Ráp vào khuôn mẫu ta có vòng lặp lồng để tính *S* như sau.

Mã 3.2.8 – Đoạn mã tính $S = 1 + x + x^2 + ... + x^n$

```
double S = 0;
for(int i = 0; i <= n; i++)
{
    double y;
    y = 1;
    for(int j = 0; j < i; j++)
        y = y * x;
    S = S + y;
}
```

Bạn hãy ráp vào hàm `main` để có chương trình hoàn chỉnh và chạy thử với *x*, *n* nào đó. Chẳng hạn với *x* = 2 thì ta có thể dùng đẳng thức $S(2, n) = 1 + 2 + 2^2 + ... + 2^n = 2^{n+1} - 1$ và xuất ra kết quả để kiểm tra thử, chẳng hạn S(2, 5) = $2^6 - 1 = 63$ (x = 2.0, n = 5 thì S = 63.0). Hay với $0 < x < 1$ thì khi *n* lớn *S*(*x*, *n*) hội tụ về $1/(1 - x)$ chẳng hạn *x* = 0.5, *n* lớn (khoảng 100) thì S $\approx 1/(1 - 0.5) = 2$.

Tuy nhiên cách viết trên thực sự không hiệu quả nghĩa là chạy rất lâu. Lưu ý thời gian chạy phụ thuộc *n* chứ không phải *x* (số lần lặp phụ thuộc *n*). Với giá trị nhỏ của *n* thì không thấy nhưng giá trị lớn sẽ cho thấy điều đó. Chẳng hạn *n* khoảng 10000 thì chạy thử trên máy của tôi tốn khoảng vài giây còn với *n* khoảng 50000 thì chắc đợi cả đời (bạn chạy và đợi thử nhé!). Có cách làm nhanh hơn không?[4] Có, tôi chỉ luôn đây!

Trước hết là cách làm "*ngây thơ*" với khuôn mẫu cũ rất dễ hiểu, dễ làm do ta đã *tách bạch* khuôn mẫu lặp bên ngoài (*i* chạy) với công việc tính số hạng của tổng (y_i, ở đây thay vì viết là *y*(*x*, *i*) thì viết là *y*(*i*) với hiểu ngầm là *y*(*i*) phụ thuộc *x*, hơn nữa *i* là biến chạy nên *y*(*i*) thường được viết là y_i). Tuy nhiên hệ quả là ta đã tách bạch việc tính các y_i, nghĩa là việc tính *y* trong các lần lặp không liên quan gì đến nhau dẫn đến ta đã tính rất nhiều. Ở đây ta nhận xét: $y_i = x^i = x^{i-1} . x = y_{i-1} . x$ với $i > 0$. Như vậy số hạng ở lần lặp *i* bằng số hạng ở lần lặp trước đó nhân thêm *x*. Vì giá trị của y_i trong các lần lặp liên quan như vậy nên ta sẽ dùng chung một biến y trong vòng lặp và thay đổi giá trị của biến này trong các lần lặp. Trong lần lặp *i*, trước khi thay đổi thì y chứa giá trị của lần lặp trước đó tức y chứa y_{i-1}. Do đó bằng cách thay đổi y = y*x (tức là $y_i = y_{i-1} . x$) ta sẽ có giá trị mới của y là y_i. Hơn nữa ta tách riêng trường hợp *i* = 0 ra (do công thức trên không dùng cho trường hợp *i* = 0). Ta

[4] Nghĩa là chương trình chạy cho ra kết quả nhanh hơn chứ không có nghĩa là bạn tốn ít thời gian để viết chương trình hơn.

có $y_0 = x_0 = 1$. Ta, như vậy, cho *y* khởi động với giá trị của y_0, *S* khởi động với giá trị của y_0, và lặp với *i* chạy từ 1 đến *n*. Do đó ta có thể viết mã như sau.

Mã 3.2.9 – Đoạn mã tính $S = 1 + x + x^2 + \ldots + x^n$ siêu siêu nhanh:)

```
double y = 1;
double S = y;
for(int i = 1; i <= n; i++)
{
    y = y * x;
    S = S + y;
}
```

Trước hết, ta nhận thấy rằng đoạn mã đã ngắn hơn và không có vòng lặp lồng. Quan trọng hơn đoạn mã này chạy siêu siêu nhanh:). Bạn thử cho n = 10000 thậm chí 50000 hay 100000 thì thời gian chạy vẫn chỉ là nháy mắt. Trong khi trước đó (Mã 3.2.8) thì thời gian chạy là vài giây, cả đời, thậm chí muôn đời nhé:) Như vậy bằng cách làm *tích hợp "tinh vi"* hơn ta đã tạo ra chương trình cực kì hiệu quả. Cách làm đầu rất dễ hiểu, dễ viết. Cách làm sau đòi hỏi phân tích, tư duy, khó hiểu hơn và khó viết hơn (cũng lưu ý là mặc dù mã ngắn hơn nhưng nó không dễ hiểu hơn đâu) bù lại là thời gian chạy nhanh hơn. Đây là một trade-off[5] giữa tính đơn giản/dễ dàng với tính hiệu quả. Quan trọng hơn hết: bạn có hiểu không? Hãy thử sức với tổng sau đây nhé:

$$S(x,n) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots \frac{x^n}{n!} = \sum_{i=0}^{n} \frac{x^i}{i!}$$

Trước hết, hãy làm theo cách ban đầu sau đó thử làm theo cách thứ hai. Gợi ý: $y_0 = 1$ và lập tỉ số $y_i/y_{i-1} = (x^i/i!)/(x^{i-1}/(i-1)!) = x/i$ nên $y_i = y_{i-1} \,.\, x/i$ với $i > 0$. Ta như vậy có đoạn mã.

Mã 3.2.10 – Đoạn mã tính $S = 1 + x + x^2/2 + x^3/6 + \ldots + x^n/n!$

```
double y = 1;
double S = y;
for(int i = 1; i <= n; i++)
{
    y = y * x / i;
    S = S + y;
}
```

Với *n* lớn thì ta có $S(x, n) \approx e^x$. Bạn dùng đoạn chương trình trên cho nhập *x* thực, *n* nguyên lớn (khoảng vài ngàn) và dùng hàm exp trong `math.h`

[5] Thuật ngữ khó có từ tiếng Việt tương ứng, hiểu nôm na là sự đánh đổi giữa cái này với cái kia, được cái này thì mất cái kia và ngược lại được cái kia thì mất cái này, ta chỉ có thể tìm một cân bằng, dung hòa hai cái.

để so sánh kết quả tính. Nếu chúng xêm xêm nhau (thậm chí là bằng nhau) thì ta làm đúng rồi.

Bạn tự mình thử sức với tổng sau đây nhé:

$$S(x,n) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots = \sum_{i=0}^{n} \frac{(-1)^i x^{2i}}{(2i)!}$$

Tổng này xấp xỉ cos*x* khi *n* lớn. Gợi ý: cũng vậy, tách riêng y_0 và lập tỉ số y_i/y_{i-1} để biết cách tính y_i từ y_{i-1} khi $i > 0$.

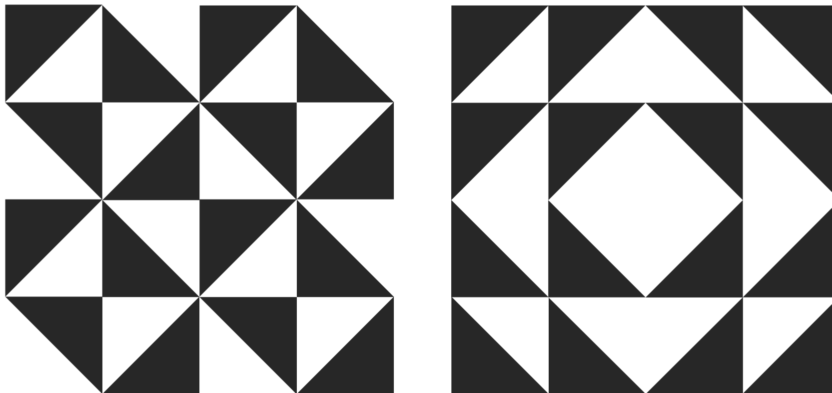
BÀI TẬP

Bt 3.2.1 Nhập *n* và xuất ra "bàn cờ vua" kích thước *n* × *n* như minh họa sau.

```
010101
101010
010101
101010
010101
010101
```

Bt 3.2.2 Nhập *n* và xuất ra các hình (kích thước phù hợp theo *n*) như minh họa sau. (Nên thử xuất ra các khối hình bộ phận đơn giản trước khi ráp lại thành hình lớn.)

Hình 3.2.1 – Các hình của Bài tập 3.2.2



Bt 3.2.3 Viết chương trình thực hiện các dạng "tính toán lặp" sau:

a) [6] $S = a + ar + ar^2 + \cdots + ar^n = \sum_{k=0}^{n} a\, r^k$

b) [7] $S = a + (a + d) + (a + 2d) + \cdots + (a + nd) = \sum_{k=0}^{n} (a + kd)$

---

[6] $S = \frac{a(1 - r^{n+1})}{1 - r}$, đây là tổng các số hạng đầu của cấp số nhân với công bội là *r* (https://en.wikipedia.org/wiki/Geometric_progression).

c) [8] $S = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$

d) [9] $\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \dots}}}}$

e) [10] $F = 1 + \cfrac{x-1}{2 + \cfrac{x-1}{2 + \cfrac{x-1}{2 + \cfrac{x-1}{2 + \dots}}}}$

f) Tính số hạng thứ *n* (*n* nguyên không âm) của dãy f_n được định nghĩa như sau[11]:

$$f_n = \begin{cases} 0 & \text{khi } n = 0 \\ 1 & \text{khi } n = 1 \\ f_{n-1} + f_{n-2} & \text{khi } n > 1 \end{cases}$$

g) Tính giá trị của *A*(*m*, *n*) (*m*, *n* nguyên không âm) được định nghĩa như sau[12]:

$$A(m,n) = \begin{cases} n+1 & \text{khi } m = 0 \\ A(m-1,1) & \text{khi } m > 0 \text{ và } n = 0 \\ A(m-1, A(m, n-1)) & \text{khi } m > 0 \text{ và } n > 0 \end{cases}$$

Bt 3.2.4 Nhập *n* và xuất ra "tam giác Pascal canh trái"[13] *n* dòng như minh họa sau. Bạn có thể xuất ra "tam giác Pascal canh giữa" không?

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
```

---

[7] $S = \frac{n(2a + nd)}{2}$, đây là tổng các số hạng đầu của cấp số cộng với công sai là *d* (https://en.wikipedia.org/wiki/Arithmetic_progression).

[8] $S \approx \sin x$, đây là tổng các số hạng đầu của khai triển Maclaurin của hàm sinx (https://en.wikipedia.org/wiki/Taylor_series).

[9] $\phi = \frac{1+\sqrt{5}}{2}$, Giá trị này được gọi là tỉ số vàng (https://en.wikipedia.org/wiki/Golden_ratio).

[10] $F = \sqrt{x}$, "Phân số lặp" như thế này (cũng như câu (d)) được gọi là liên phân số (https://en.wikipedia.org/wiki/Continued_fraction).

[11] Cách định nghĩa này gọi là đệ qui (mà ta sẽ tìm hiểu kĩ sau). Dãy f_n này được gọi là dãy Fibonacci (https://en.wikipedia.org/wiki/Fibonacci_number).

[12] *A* được gọi là hàm Ackermann. Lưu ý là *A* lớn rất nhanh, do đó chỉ thử tính với (*m*, *n*) rất nhỏ (https://en.wikipedia.org/wiki/Ackermann_function).

[13] Left-justified Pascal's triangle (https://en.wikipedia.org/wiki/Pascal%27s_triangle).

BÀI 3.3

Bài này bắt đầu bằng lưu đồ Hình 2.3.2 của Bài 2.3. Ta hãy viết mã C cho lưu đồ này.

Mã 3.3.1 – Khung chương trình tìm nghiệm đa thức $ax^2 + bx + c$

```
#include <stdio.h>
int main()
{
    double a, b, c;

    printf("Nhap cac he so a, b, c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    if(a != 0)
    {
        // tìm nghiệm của đt bậc 2: ax^2+bx+c (a ≠ 0)
    }
    else
    {
        // tìm nghiệm của đt bậc 1: bx + c
    }

    return 0;
}
```

Hiển nhiên là lưu đồ trên chưa chạy được vì có chứa 2 nút *hộp đen*. Tương ứng thì đoạn mã C trên chưa chạy được. Ta cần *phân giải* 2 nút hộp đen trong lưu đồ, tương ứng thì ta cần viết mã cho 2 chỗ ghi chú trên. Ta phân giải nút hộp đen bằng cách viết lưu đồ cho công việc tương ứng của nút hộp đen (mà ta gọi là *lưu đồ con*) và "ghép" nó vào lưu đồ lớn (mà ta gọi là *lưu đồ cha*) bằng cách chèn mã tương ứng cho các lưu đồ vào mã cho lưu đồ lớn (chèn vào vị trí ghi chú ở mã trên). Đây là cách mà ta đã làm cho đến giờ. Tuy nhiên, có một lựa chọn khác đó là để riêng mã của lưu đồ con và "gọi" mã đó trong mã của lưu đồ cha. Hiển nhiên câu hỏi trước hết là cách nào tốt hơn, "gọi" hay "chèn"? Hay khi nào nên "gọi" và khi nào nên "chèn"?

Trước khi trả lời câu hỏi đó, ta xem cách thức làm điều này. Trong C, mã của lưu đồ con được để riêng trong cái gọi là *hàm của C*. Việc mô tả được gọi là *viết hàm* hay *định nghĩa hàm*. Bạn hãy viết hàm sau lên trước hàm `main`, đây chính là mã của lưu đồ giải phương trình bậc 2 (Hình 2.3.1) được để riêng.

Mã 3.3.2 – Hàm cho lưu đồ con "giải phương trình bậc 2"

```
void giaidtbac2(double a, double b, double c)
{
    double delta = b*b - 4*a*c;
    if(delta < 0)
        printf("Da thuc vo nghiem.\n");
    else if(delta == 0)
        printf("Da thuc co 1 nghiem: %lf.\n", -b/(2*a));
    else // delta > 0
    {
        double x1 = (-b - sqrt(delta))/(2*a);
        double x2 = (-b + sqrt(delta))/(2*a);
        printf("Da thuc co 2 nghiem: %lf, %lf.\n",
             x1, x2);
    }
}

// hàm main ở đây
```

Việc gọi chạy lưu đồ con để riêng được thực hiện bằng *lời gọi hàm*. Tại chỗ ghi chú trong hàm `main` tương ứng với nút hộp đen giải phương trình bậc hai, bạn hãy thay bằng lời gọi hàm sau đây nhé:

```
giaidtbac2(a, b, c);
```

Hiển nhiên đây là lệnh đơn nên không cần cặp ngoặc `{}` trong thân `if`. Nghĩa là có thể viết như đoạn mã sau:

```
if(a != 0)
    giaidtbac2(a, b, c);
else
{
    // tìm nghiệm của đt bậc 1: bx + c
}
```

Bạn đã có thể chạy chương trình trên (nhớ `#include <math.h>` vì ta có dùng hàm `sqrt`), nhớ nhập giá trị a khác 0 để chương trình chạy theo hướng giải phương trình bậc 2 (hướng giải phương trình bậc nhất khi a bằng 0 ta sẽ làm tương tự sau).

Điều quan trọng đầu tiên là ta phải phân biệt được định nghĩa của hàm với lời gọi hàm. *Định nghĩa của hàm* là mã của lưu đồ con được viết riêng ra. *Lời gọi hàm* giúp ta móc lưu đồ con vào lưu đồ cha mà không chèn trực tiếp. Thứ hai, để có thể gọi hàm ta cần đặt tên cho hàm, và gọi hàm bằng cách dùng *tên hàm* (cũng như ta truy cập biến bằng tên biến). Chẳng hạn hàm trên có tên là `giaidtbac2`. Tên hàm được chọn khi ta định nghĩa hàm, sau đó ta cần dùng đúng tên này để gọi thực thi hàm tương ứng (tức là chạy lưu đồ con tương ứng). Về mặt từ vựng tên hàm là một định danh (có qui tắc như tên biến), và cũng như tên biến, ta nên đặt tên hàm ngắn gọn và gợi

nhớ công việc của hàm, thường thì đó là một động từ (có thể có bổ ngữ kèm theo)[1]. Chẳng hạn tên hàm cho việc giải đa thức bậc 2 có thể đặt là `giaidtbac2` như trên. Thứ ba, hàm có thể có *tham số*. Những công việc nhỏ ứng với các lưu đồ con thường là công việc được tham số (xem Bài 3.2). Chẳng hạn việc giải đa thức bậc hai có tham số chính là đa thức cần giải, cụ thể chính là 3 hệ số *a*, *b*, *c* (một đa thức bậc 2 được xác định duy nhất bởi 3 hệ số). 3 tham số này còn được gọi là *đầu vào* (input) của hàm vì 3 tham số này nhận giá trị trước khi hàm bắt đầu thực thi. Khi công việc của hàm có tham số thì trong định nghĩa hàm ta cần xác định các tham số này, mỗi tham số cần xác định kiểu và tên của tham số. Cũng vậy tên của tham số là một định danh và cần đặt gợi nhớ. Chẳng hạn hàm trên có 3 tham số chính là 3 hệ số, tất cả đều có kiểu `double` và tên là `a`, `b`, `c` (theo cách Toán hay đặt cho 3 hệ số của đa thức bậc 2). Khuôn mẫu một hàm như vậy là:

```
void <tên hàm>(<các tham số>)
{
    <công việc của hàm>
}
```

Nếu không có tham số thì ta để trống (dĩ nhiên là vẫn có cặp ngoặc tròn bên ngoài). Các tham số được khai báo theo khuôn mẫu:

```
<kiểu tham số> <tên tham số>
```

Nếu có nhiều tham số thì dùng dấu phẩy (`,`) để phân cách các tham số. Bạn xem định nghĩa hàm `giaidtbac2` để hiểu rõ hơn nhé.

Ở trên ta thấy công việc của hàm có cú pháp là một khối lệnh (`{ … }`). Ta gọi khối lệnh này là *thân hàm*. Mọi điều ta đã biết về khối lệnh như vậy có thể được áp dụng ở đây. Chẳng hạn nó có thể là khối lệnh rỗng. Khi đó ta có một hàm "không làm gì cả"[2].

Điều quan trọng là ta cần phân biệt tham số với giá trị truyền cho tham số. Tham số được xác định trong định nghĩa hàm còn giá trị truyền cho tham số được cho trong lời gọi hàm. Chẳng hạn hàm `giaidtbac2` có 3 tham số là `a`, `b`, `c` (xem định nghĩa hàm). Để giải đa thức: $x^2 + 2x + 1$, ta cần giải đa thức $ax^2 + bx + c$ với $a = 1$, $b = 2$, $c = 1$. Như vậy ta gọi hàm với lời gọi:

```
giaidtbac2(1, 2, 1)
```

để giải đa thức bậc hai cụ thể trên. Tương tự để giải đa thức bậc hai: $x^2 + x$, ta có thể gọi: `giaidtbac2(1, 1, 0)`. Như vậy cú pháp chung cho lời gọi hàm là:

```
<tên hàm>(<đối số 1>, <đối số 2>, …, <đối số n>)
```

---

[1] Khác với tên biến thường là danh từ.

[2] Một hàm như vậy thì không có ích gì nhưng là hợp lệ.

<tên hàm> là tên của hàm cần gọi (tên hàm cần thực thi, chính là tên của lưu đồ con cần chạy). Các *đối số* là các biểu thức. Số lượng đối số phải bằng số lượng tham số trong định nghĩa hàm. Nếu không có thì để trống, nếu có nhiều thì dùng dấu phẩy phân cách như trên. Khi thực hiện lời gọi hàm, các đối số, là các biểu thức sẽ được lượng giá[3]. Sau đó các giá trị kết quả sẽ được gán cho các tham số tương ứng của hàm và thân hàm được *thực thi*. Việc gán này còn được gọi là *truyền giá trị cho tham số*. Khi thực thi thân hàm thì có thể xem các tham số là các biến đã được gán giá trị trước, là giá trị của các đối số. Ở đây như vậy C sẽ tự động gán giá trị của các đối số cho các tham số trước khi thực thi. Giá trị đối số nào sẽ được gán cho tham số nào? C sẽ dùng thứ tự tương ứng của các đối số và tham số. Đối số thứ nhất được gán cho tham số thứ nhất, đối số thứ hai được gán cho tham số thứ hai, …, đối số cuối cùng được gán cho tham số cuối cùng. Như vậy mặc dù ta được tùy ý lựa chọn thứ tự cho các tham số trong định nghĩa hàm. Nhưng khi gọi hàm thì ta phải truyền đối số theo đúng thứ tự với tham số tương ứng. Một điều nữa, giá trị của tham số phải có kiểu phù hợp với kiểu của tham số. Giống kiểu là tốt nhất (giá trị thực 1.0 được truyền cho tham số kiểu `double`) hoặc là *phù hợp kiểu* và khi đó có *chuyển kiểu*. Chẳng hạn lời gọi `giaidtbac2(1, 2, 1)` thì các đối số `1`, `2`, `1` là các số nguyên sẽ được tự động chuyển kiểu thành các giá trị `1.0`, `2.0`, `1.0` thực trước khi gán cho các tham số `a`, `b`, `c` tương ứng theo thứ tự do các tham số này có kiểu là `double`.

Hãy xem lời gọi hàm trong đoạn mã trên:

```
giaidtbac2(a, b, c)
```

Ta lưu ý là các đối số là các biểu thức và trường hợp đơn giản thì đó là các biểu thức đơn như các hằng trong lời gọi `giaidtbac2(1, 2, 1)`, còn ở đây là các biến. Là gì đi nữa thì giá trị của nó sẽ được tính sau đó được gán vào các tham số. Điều nhập nhằng ở đây là có sự trùng tên các kí hiệu. Cụ thể là ta có các biến nhập `a`, `b`, `c` của hàm `main` (được khai báo tại Dòng 4 Mã 3.3.1) và các tham số trùng tên `a`, `b`, `c` của hàm `giaidtbac2` (được khai báo trong định nghĩa hàm tại Dòng 1 Mã 3.3.2). Tuy nhiên ta cần phân biệt rõ ràng. Trong thân hàm `main` (bao gồm lời gọi hàm `giaidtbac2`) thì `a`, `b`, `c` là 3 biến nhập, được khai báo tại Dòng 4 và có giá trị do người dùng nhập vào từ lệnh nhập Dòng 7. Trong định nghĩa hàm `giaidtbac2` thì `a`, `b`, `c` là 3 tham số của hàm. Bọn chúng là khác nhau. Bọn chúng chỉ tình cờ trùng tên nhau (do ta thường đặt tên các hệ số là *a*, *b*, *c*). Bọn chúng có liên quan qua việc truyền giá trị cho tham số trong lời gọi hàm. Giá trị của `a`, `b`, `c` (là 3 biến nhập của hàm `main`) được tính (ở đây là lấy giá trị trong biến tương ứng) sau đó giá trị được gán cho 3 tham số `a`, `b`, `c` của hàm `giaidtbac2`. Khi thân

[3] Thứ tự lượng giá các đối số tùy thuộc vào compiler nhưng thường là theo thứ tự liệt kê các đối số.

hàm `giaidtbac2` chạy thì `a`, `b`, `c` trong thân hàm (chẳng hạn tại dòng tính `delta`, Dòng 3 Mã 3.3.2) là các tham số (với giá trị đã được gán trước đó).

Tương tự như trên ta định nghĩa hàm `giaidtbac1` cho công việc tìm nghiệm khi $a = 0$. Một đa thức bậc nhất là đa thức có dạng $ax + b$[4]. Lại lưu ý việc nhập nhằng kí hiệu, a, b ở đây không phải là a, b trong bài toán $ax^2 + bx + c$ khi $a = 0$. Vấn đề là đa thức bậc nhất được xác định duy nhất bởi hai hệ số: hệ số đi với x và hệ số tự do. Theo thói quen, cái đầu đặt là a, cái sau là b. Ta viết hàm giải đa thức $ax + b$ như sau:

Mã 3.3.3 – Hàm cho lưu đồ con "giải đa thức $ax + b$"

```
void giaidtbac1(double a, double b)
{
    if(a == 0)
    {
        if(b == 0)
            printf("Da thuc co vo so nghiem");
        else
            printf("Da thuc vo nghiem");
    }
    else
        printf("Da thuc co 1 nghiem: %lf.\n", -b/a);
}
```

Và trong `main` ta gọi hàm `giaidtbac1` như sau:

```
if(a != 0)
    giaidtbac2(a, b, c);
else
    giaidtbac1(b, c);
```

Như vậy nếu `a` (của `main`) khác 0 thì đó là trường hợp đa thức bậc hai nên ta gọi `giaidtbac2` để giải. Trường hợp `a` bằng 0 (`else`) thì ta gọi `giaidtbac1` nhưng đa thức bậc 1 khi đó là `b`x + `c` (với `b`, `c` của `main`). Nhưng trong định nghĩa của `giaidtbac1` thì hệ số gắn với x được đặt tên là `a`, còn hệ số tự do đặt tên là `b`, nghĩa là ở `giaidtbac1` đi giải đa thức `a`x + `b` (`a`, `b` là tham số của `giaidtbac1`). Theo thứ tự khai báo tham số thì `a` để trước `b` (`a` thứ nhất, `b` thứ nhì). Nên lời gọi hàm là `giaidtbac1(b, c)` (`b`, `c` của `main`). Tôi không chắc bạn có rõ ràng được cái lằng nhằng này không. Tôi hết cách nói rồi (nói nữa sẽ làm rối thôi). Nếu chưa rõ, bạn tự mình nghiền ngẫm nhé[5]. Nếu bạn rõ chỗ này thì bạn đã phân biệt được tham số với đối số. Chúc mừng, công lực của bạn đã tăng cao một chút!

[4] Chặt chẽ hơn thì khi $a \neq 0$ thì $ax + b$ mới được gọi là đa thức bậc nhất. Ở đây ta cứ gọi bừa:)

[5] Có thể dừng lại giải lao và suy nghĩ một chút.

Tại sao có rắc rối này. Đó là vấn đề *ngữ cảnh*. Bạn nghĩ chữ *a* có nghĩa là gì (nó không chỉ là chữ cái đầu tiên, nó còn là mạo từ *a*, và nó được dùng thường xuyên trong Toán với ý nghĩa khác nhau trong các ngữ cảnh khác nhau). Ở đây, rất đơn giản khi trong `main` (lời gọi hàm nằm trong `main`) thì `a`, `b`, `c` là của `main`. Khi qua hàm `giaidtbac1` thì `a`, `b` là của đa thức bậc nhất. Hệ số đi với *x* trong `main` thì gọi là `b`, còn trong `giaidtbac1` thì nó lại được gọi là `a`[6].

Cuối cùng ta cần thấy vai trò của hàm `main`. Mọi lưu đồ lớn nhỏ đều viết thành hàm trong C. Lưu đồ lớn nhất hay *lưu đồ chính* được viết trong hàm có tên là `main`. Bạn phải dùng đúng cái tên này cho lưu đồ chính, tức là lưu đồ của chương trình. Các lưu đồ con khác để trong các hàm khác. Ta có đúng một hàm `main` ứng với đúng một lưu đồ chính. Nếu ta có thêm các lưu đồ con thì sẽ có thêm các hàm khác. Nếu công việc của một lưu đồ có dùng một lưu đồ con khác thì trong thân hàm của lưu đồ cha sẽ có lời gọi đến hàm ứng với lưu đồ con. Khi đó ta gọi hàm ứng với lưu đồ con là *hàm được gọi* (*callee*) còn hàm ứng với lưu đồ cha là *hàm gọi* (*caller*).

Một điều nữa về mặt kĩ thuật: nếu hàm *A* gọi hàm *B* (tức *A* là caller và *B* là callee) thì định nghĩa hàm *B* phải đặt trước định nghĩa hàm *A* (theo nguyên tắc: phải có trước khi dùng). Đó là lí do mà định nghĩa của hàm `giaidtbac2` và `giaidtbac1` phải đặt trước định nghĩa của hàm `main` do trong `main` có gọi hai hàm này. Thứ tự của `giaidtbac2` với `giaidtbac1` là tùy ý (nghĩa là `giaidtbac2` đặt trước `giaidtbac1` hoặc `giaidtbac1` đặt trước `giaidtbac2` đều được) vì hai hàm này không liên quan (`giaidtbac1` không gọi `giaidtbac2` và `giaidtbac2` cũng không gọi `giaidtbac1`).

Một cách khác là ta có thể tách riêng *khai báo hàm* khỏi định nghĩa hàm. Khai báo hàm chính là dòng đầu của định nghĩa hàm (kết thúc bằng dấu `;`), còn gọi là *prototype* của hàm, nghĩa là:

```
void <tên hàm>(<các tham số>);
```

Chẳng hạn prototype của hàm `giaidtbac2`:

```
void giaidtbac2(double a, double b, double c);
```

Khai báo hàm cho biết có một hàm có tên với các đối số như vậy. Định nghĩa hàm thì phải có thêm phần cài đặt cho thân hàm. Thật ra ta có thể không cần tên tham số trong khai báo hàm mà chỉ cần kiểu thôi vì ta không dùng đến tham số. Hiển nhiên là cần có tên tham số trong định nghĩa hàm vì trong thân hàm ta có dùng đến các tham số. Như vậy khai báo của hàm trên có thể được viết gọn hơn là:

```
void giaidtbac2(double, double, double);
```

[6] Kì cục một cách tự nhiên:)

Khi hàm *A* có gọi hàm *B* thì chỉ cần khai báo của hàm *B* đặt trước định nghĩa của hàm *A* là được, còn định nghĩa của hàm *A* có thể đặt trước hay sau *B* đều được[7]. Như vậy toàn bộ chương trình trên có thể viết lại như sau và đây là cách mà người ta hay viết.

Hình 3.3.1 – Tổ chức chương trình tìm nghiệm đa thức $ax^2 + bx + c$ (minh họa cách viết chương trình có nhiều hàm)

```
#include <stdio.h>
#include <math.h>

void giaidtbac2(double, double, double);
void giaidtbac1(double, double);

int main()
{
    ...
}

void giaidtbac2(double a, double b, double c)
{
    ...
}

void giaidtbac1(double a, double b)
{
    ...
}
```

MỞ RỘNG 3.3 – Hàm gọi lẫn nhau

Bạn đã biết thế nào là một số nguyên chẵn: là số chia hết cho 2. Ngược lại số nguyên không chia hết cho 2 gọi là số lẻ. Thế có cách nào khác để định nghĩa (hay xác định) tính chẵn/lẻ của một số nguyên không? Nhận xét rằng một số nguyên chỉ có thể hoặc chẵn hoặc lẻ (mà không thể vừa chẵn vừa lẻ cũng như không thể không chẵn cũng không lẻ) và tính chẵn/lẻ luân phiên trên các số nguyên (0 chẵn, 1 lẻ, 2 chẵn, 3 lẻ, …). Từ đó ta có cách định nghĩa tính chẵn/lẻ của một số nguyên không âm *n* như sau:

(1) Nếu $n = 0$ thì *n* chẵn và do đó không lẻ.

(2) Nếu $n > 0$ thì:

   (a) *n* chẵn nếu n – 1 lẻ.

   (b) *n* lẻ nếu n – 1 chẵn.

Định nghĩa trên trông có vẻ lẩn quẩn vì chẵn được định nghĩa bằng lẻ

[7] Thậm chí có thể đặt ở một file khác như ta sẽ thấy sau.

mà lẻ lại được nghĩa bằng chẵn. Tuy nhiên đó là một định nghĩa tốt. Thật vậy: từ (1) ta có 0 chẵn, từ (2.b) và do 0 chẵn nên 1 lẻ, từ (2.a) và do 1 lẻ nên 2 chẵn, … Cách định nghĩa này được gọi là *đệ qui tương hỗ* mà ta sẽ tìm hiểu kĩ sau. Ở đây ta sẽ viết hai hàm trong C: hàm even và odd để kiểm tra một số nguyên không âm n có chẵn hay lẻ không.

Mã 3.3.4 – Một ví dụ hàm gọi lẫn nhau (đệ qui tương hỗ) trong C

```
1  int even(int);
2  int odd(int);
3
4  int even(int n)
5  {
6      return n == 0 ? 1 : odd(n - 1);
7  }
8
9  int odd(int n)
10 {
11     return n == 0 ? 0 : even(n - 1);
12 }
```

Hàm even (hay odd) sẽ trả về đúng (1) khi tham số n chẵn (hay lẻ) và trả về sai (0) khi tham số n không chẵn (hay không lẻ). Ngoài việc dùng số nguyên cho giá trị luận lý (0 là sai, 1 là đúng) và biểu thức điều kiện (xem lại Bài 2.6) thì cài đặt của even/odd hoàn toàn tuân theo định nghĩa đệ qui của tính chẵn/lẻ ở trên. Tính lẩn quẩn của định nghĩa thể hiện qua việc hàm even gọi hàm odd và hàm odd gọi lại hàm even. Ta nói even và odd gọi lẫn nhau.

Nếu không có các khai báo hàm ở Dòng 1 và Dòng 2 thì C sẽ báo lỗi ở lời gọi đến hàm odd (Dòng 6) trong thân hàm even vì C không thấy odd trước đó. Lưu ý là lời gọi đến hàm even (Dòng 11) trong thân hàm odd không có vấn đề gì vì hàm even đã xuất hiện trước (trên) hàm odd. Nếu ta đổi định nghĩa hàm odd lên trước định nghĩa hàm even thì ta lại bị lỗi ở lời gọi đến hàm even trong thân hàm odd (lời gọi đến hàm odd trong thân hàm even không có vấn đề gì). Đây là một vấn đề trong C mà ta có thể giải quyết bằng cách dùng *khai báo hàm*[8] như mã trên. Thật ra trong mã trên ta chỉ cần khai báo trước hàm odd là đủ (nghĩa là cần Dòng 2 nhưng không cần Dòng 1). Tuy nhiên để cho “cân đối” và “dự phòng” trường hợp định nghĩa hàm odd được đưa lên trước định nghĩa hàm even thì ta nên có đầy đủ khai báo như mã trên.

Tóm lại khi viết chương trình có nhiều hàm, ta nên tuân theo khuôn mẫu như Hình 3.3.1.

[8] Với một số ngôn ngữ lập trình như C#, Java, … thì không có vấn đề gì cả với định nghĩa của các hàm gọi lẫn nhau.

BÀI TẬP

Bt 3.3.1 Viết hàm xác định số lớn nhất trong hai số. Sau đó dùng hàm này để tìm số lớn nhất trong *n* số. (*Gợi ý*: max{*a*, *b*, *c*} = max{max{*a*, *b*}, *c*}.)

Bt 3.3.2 Viết hàm tìm ước số chung lớn nhất của hai số nguyên dương. Sau đó dùng hàm này để tìm ước số chung lớn nhất trong *n* số nguyên dương. (*Gợi ý*: tương tự Bài tập 3.3.1.)

Bt 3.3.3 Viết hàm tính số tổ hợp $C_n^k = \frac{n!}{k!(n-k)!} (n \geq 0, 0 \leq k \leq n)$ bằng cách "tách bạch, ngây thơ" và cách "tích hợp, tinh vi" (xem Bài 3.2):

- Viết riêng hàm tính giai thừa và dùng hàm này khi tính C_n^k.
- Phân tích, biến đổi, … để tính C_n^k.

So sánh ưu khuyến điểm của hai cách làm.

Bt 3.3.4 Viết hàm nhận một kí tự chữ cái tiếng Anh (a, ..., z, A, ..., Z) và xuất ra "chữ viết đẹp" cho chữ cái đó. Chẳng hạn ở Bài tập 1.1.4 là kết xuất cho chữ cái C[9].

Bt 3.3.5 Làm lại Bài tập 2.4.6 (và Bài tập 3.1.5) bằng cách tổ chức hàm phù hợp.

Bt 3.3.6 Làm lại Bài tập 2.5.4 bằng cách tổ chức hàm phù hợp.

Bt 3.3.7 Làm lại Bài tập 2.5.5 bằng cách tổ chức hàm phù hợp.

Bt 3.3.8 Hãy tổ chức chương trình cho 2 hàm gọi lẫn nhau `f1`, `f2` sau để hàm `main` có thể gọi `f1` hay `f2`. Kết quả của lời gọi `f1(10)` là gì? của `f2(10)` là gì? Bạn có biết công việc của `f1`, `f2` là gì không?

```
void f1(int n)
{
    if(n > 0)
    {
        printf("0");
        f2(n - 1);
    }
}
void f2(int n)
{
    if(n > 0)
    {
        printf("1");
        f1(n - 1);
    }
}
```

[9] Xem Bài 2.7 để biết thêm về "thiết kế font chữ".

BÀI 3.4

Trong Bài 3.2 ta đã xuất các hình khác nhau bằng cách chèn mã cho công việc “Xuất các kí tự ở dòng thứ *i*”. Trong bài này ta sẽ làm theo kiểu “gọi” chứ không phải theo kiểu “chèn”. Nghĩa là ta sẽ viết hàm cho các công việc đó. Công việc “Xuất các kí tự ở dòng thứ i” (Dòng 3 Mã 3.2.3) nói chung là *công việc được tham số hóa* theo *i* (dòng thứ mấy) và *n* (tổng số dòng) cho nên prototype của nó là: `void xuatdong(int d, int n)`. Ở đây ta đặt lại tên tham số dòng thứ mấy là `d` (viết tắt của dòng) vì tên `i` thường dùng cho biến lặp. Dĩ nhiên khi gọi thì bạn phải truyền đối số phù hợp (chẳng hạn là `xuatdong(i, n)` với `i` là biến lặp của nơi gọi).

Trong trường hợp đầu tiên, xuất ra một dấu sao trên mỗi dòng, thì hàm `xuatdong` được viết là:

Mã 3.4.1 – Hàm xuatdong cho xuất ra một dấu sao trên mỗi dòng

```
void xuatdong(int d, int n)
{
    printf("*");
}
```

Đầu tiên ta để ý là mặc dù nói chung việc xuất các kí tự ở dòng thứ *d* là phụ thuộc vào vị trí dòng và tổng số dòng (tức là phụ thuộc tham số *d* và *n*). Nhưng trường hợp đặc biệt này thì không cần. Bạn có thể viết như trên hoặc có thể viết gọn hơn bằng:

Mã 3.4.2 – Hàm xuatdong viết gọn hơn

```
void xuatdong()
{
    printf("*");
}
```

Khi đó lời gọi hàm cũng sẽ đơn giản là `xuatdong()` mà không có đối số. Câu hỏi ở đây là: lựa chọn viết hàm cho công việc con (để riêng lưu đồ con) hay viết mã trực tiếp (chèn lưu đồ con) là tốt hơn? Trong trường hợp này công việc con quá đơn giản, chỉ gồm một lệnh thì rõ ràng là không đáng để ta viết riêng hàm. Khi viết riêng hàm thì vừa rườm rà về mã (viết riêng hàm ra và có thể đặt prototype trước nơi gọi, rồi gọi hàm) và cũng kém hiệu quả hơn khi thực thi (để thực thi một lời gọi hàm thì tốn nhiều thời gian và cả bộ nhớ). Nói chung khi công việc con là đơn giản (có thể viết thành một hoặc một vài lệnh đơn giản) thì ta nên chèn trực tiếp mã vào thay vì dùng hàm để viết riêng ra.

Trường hợp thứ 2 (xuất ra mỗi dòng *n* dấu * để được hình vuông) có thể viết hàm xuatdong như sau:

Mã 3.4.3 – Hàm xuatdong cho xuất ra *n* dấu sao trên mỗi dòng

```
void xuatdong(int n)
{
    for(int j = 1; j <= n; j++)
        printf("*");
}
```

Trường hợp xuất ra hình tam giác vuông cân hay 3 hình tam giác đều thì sao? Bạn hãy viết hàm xuatdong phù hợp và ráp vào hàm main với lời gọi hàm phù hợp để chương trình xuất ra các loại hình nhé. Theo bạn thì nên viết hàm hay chèn mã trực tiếp? Tùy mỗi người thôi. Với tôi, tôi sẽ viết mã trực tiếp trong trường hợp này vì mã khá đơn giản.

Tiếp theo ta cũng có công việc được tham số hóa "Tính giá trị $y(x, i)$" với $y(x, i) = x^i$ tức là công việc tính lũy thừa x^i (xem lại phần tính tổng hữu hạn trong Bài 3.2). Điểm khác biệt so với tất cả các công việc được viết riêng tới giờ là công việc này cần tính ra một số, ta có thể làm nhiều bước như các công việc khác, nhưng cuối cùng phải đưa ra được một số (là x^i trong trường hợp này). Ta nói công việc này có *trả về giá trị* mà trong trường hợp này đó là một giá trị thực (kiểu double). Khi đó prototype của hàm này có thể là:

```
double luythua(double x, int n);
```

Vì mục đích là tính lũy thừa nên ta đặt tên hàm là luythua. Công việc được tham số theo số cần lũy thừa (x, là số thực bất kì nên kiểu double) và số mũ (ở đây số mũ là số nguyên không âm (kiểu int), và trong Toán hay kí hiệu là *n* nên ta đặt tên là n). Điều quan trọng là từ khóa double thay cho từ khóa void như trước. Điều này báo cho biết hàm này trả về một giá trị thực (kiểu double). Như vậy nếu công việc con không trả về giá trị thì ta dùng void còn khi trả về giá trị thì ta cho biết kiểu của giá trị trả về. Hàm không trả về giá trị còn được gọi là *hàm void* hoặc là *thủ tục*. Còn hàm trả về giá trị là *hàm thông thường*. Trong C, cả hai đều được gọi là hàm.

Hàm luythua có thể được viết như sau (dựa trên Bài 3.2).

Mã 3.4.4 – Hàm tính lũy thừa x^n

```
double luythua(double x, int n)
{
    double y = 1;
    for(int i = 0; i < n; j++)
        y = y * x;

    return y;
}
```

Đối chiếu với đoạn Mã 3.2.8. Đầu tiên ta có phải khai báo kiểu cho y và biến lặp được thay bằng `i` thay vì `j`. Lưu ý là y hay biến lặp này không liên quan gì đến y hay biến lặp ở `main`. Một cách tổng quát thì các biến trong một hàm không liên quan đến các biến trong hàm khác. Điều này có nghĩa: mã của các hàm là độc lập nhau. Nhờ đó mà việc viết mã cho hàm dễ dàng hơn, bạn chỉ cần tập trung vào hàm đang viết mà không cần để ý đến các hàm khác. Chẳng hạn, bạn có thể dùng các kí hiệu (tên biến) tùy ý mà không sợ trùng kí hiệu (tên biến) với các hàm khác. Như với biến lặp ta hay dùng tên là `i` (có `i` rồi mới đến `j`, `k`). Ở đây ta dùng `i` mà không sợ trùng với `i` của hàm `main` (chúng không liên quan gì). Dĩ nhiên là các hàm liên quan đến nhau qua lời gọi hàm.

Điều quan trọng là để ý đến lệnh `return` ở Dòng 7 trên. Khi hàm có trả về giá trị thì bạn phải dùng lệnh `return` để trả về giá trị (mong đợi) cho hàm. Cú pháp lệnh này là:

return <biểu thức *E*>;

trong đó `return` là từ khóa. Khi gặp lệnh này thì *E* sẽ được lượng giá, hàm sẽ kết thúc và giá trị trả về chính là giá trị của *E*. Lưu ý là giá trị của *E* phải có kiểu phù hợp với kiểu được khai báo trong prototype của hàm (ở đây là `double`), và có thể có chuyển kiểu nếu cần. Lệnh này (còn được gọi là lệnh `return`) thuộc nhóm *lệnh nhảy* (xem lại Bài 2.6) vì khi gặp lệnh này, hàm sẽ kết thúc thực hiện dù còn các lệnh khác sau đó trong thân hàm. Trường hợp hàm không có giá trị trả về (hàm `void` hay thủ tục) thì dĩ nhiên ta không cần dùng lệnh `return` nhưng ta cũng có thể dùng nó với cú pháp:

```
return;
```

Lưu ý là không có biểu thức đi kèm. Dạng này được gọi là `return` *không đối số*. Khi đụng lệnh này, thủ tục sẽ kết thúc ngay (không có giá trị trả về). Hiển nhiên nếu bạn đã báo hàm có giá trị trả về (hàm thường) thì phải dùng `return` *có đối số* còn hàm `void` thì nếu có dùng thì là `return` không đối số. Bằng ngược lại sẽ là lỗi ngữ nghĩa.

Khi đã có hàm `luythua` trên thì đoạn Mã 3.2.8 có thể được viết lại là:

Mã 3.4.5 – Mã 3.2.8 viết lại bằng cách dùng hàm `luythua`

```
double S = 0;
for(int i = 0; i <= n; i++)
{
    double y;
    y = luythua(x, i);
    S = S + y;
}
```

Hoặc viết gọn là:

Mã 3.4.6 – Mã 3.4.5 viết gọn

```
double S = 0;
for(int i = 0; i <= n; i++)
    S = S + luythua(x, i);
```

Khi đó `luythua(x, i)` có vai trò như là một biểu thức. Thật vậy *lời gọi hàm là một biểu thức*, các toán hạng (các biểu thức con) là các đối số và giá trị trả về của hàm là giá trị của biểu thức. Lời gọi đến hàm `void` (thủ tục) cũng là một biểu thức nhưng là *biểu thức không có giá trị* (xem lại Bài 2.1). Vì lời gọi đến hàm void là một biểu thức không có giá trị nên nó chỉ có ý nghĩa khi nó có hiệu ứng lề, nghĩa là nó gây nên những thay đổi nào đó (như nhập hay xuất cái gì đó, …) hay nói chung là làm gì đó nên nó được gọi là thủ tục. Một hàm thông thường (có giá trị trả về) thì thường là tính toán gì đó và thường không có hiệu ứng lề. Tuy nhiên C không phân biệt rạch ròi những trường hợp này: chúng đều được gọi là hàm của C và một hàm thường cũng có thể có hiệu ứng lề và hàm `void` cũng có thể không có hiệu ứng lề.

Câu hỏi là có nên viết riêng hàm như vậy không? Câu trả lời là có, vì mã cũng không đơn giản, hàm này cũng có ý nghĩa độc lập và cũng có thể hay được dùng (hàm lũy thừa mà lị[1]). Tuy nhiên nếu bạn chọn cách làm này thì bạn sẽ không dùng được cách "tích hợp" (so với cách "phân tách") như đoạn Mã 3.2.9.

Đoạn Mã 3.2.10 tính $S(x, n)$. Giá trị này xấp xỉ e^x khi n lớn. Đây là một hàm Toán quan trọng nên ta có thể viết tách riêng ra thành hàm như sau.

Mã 3.4.7 – Hàm tính lũy thừa e^x

```
double exp(double x)
{
    double y = 1;
    double S = y;
    for(int i = 1; i <= 10000; i++)
    {
        y = y * x / i;
        S = S + y;
    }

    return S;
}
```

Thật sự thì hàm này quan trọng đến nỗi mà nó có trong thư viện chuẩn của C (cùng tên trong `math.h`). Tuy nhiên có phải người ta viết hàm đó như ta đã viết không? Hay có cách nào làm hiệu quả hơn (nhanh hơn và chính xác hơn) không?

[1] Thư viện chuẩn cũng có hàm tính lũy thừa (hàm `pow` trong `math.h`) nhưng tính x^y tổng quát hơn (x, y đều là các số thực).

Như vậy cho đến lúc này: "Hàm là khối lệnh được đặt tên, có thể được tham số hóa (input) và có thể có kết quả trả về (output). Hàm có thể được gọi thực thi với input phù hợp (đối số) và có thể nhận giá trị trả về khi kết thúc."

Ta đã thấy trường hợp hàm không có giá trị trả về (hàm `void`) và hàm có một giá trị trả về. Trường hợp ta muốn hàm trả về nhiều giá trị[2] thì sao? Cho một số thực, ta gọi phần nguyên là phần trước dấu chấm thập phân còn phần lẻ là phần sau dấu chấm. Chẳng hạn số 1.5 có phần nguyên là 1 còn phần lẻ là 0.5. Dĩ nhiên phần nguyên là số nguyên còn phần lẻ là số thực. Giả sử ta cần viết hàm `f` nhận một tham số là số thực `x`, `f` trả về phần nguyên và phần lẻ của `x`. Ta có thể viết hàm này như sau.

Mã 3.4.8 – Hàm tính phần nguyên và phần lẻ của một số thực

```
int f(double x, double &rem)
{
    int d = (int)x;
    rem = x - d;

    return d;
}
```

Ta sẽ cho hàm `f` trả về phần nguyên (kiểu `int`). Ngoài ra hàm `f` có thêm một tham số đặc biệt `rem` chứa phần lẻ (số thực - kiểu `double`). Về ý nghĩa thì tham số `rem` khác với tham số bình thường `x` là nó dùng để chứa giá trị trả về của hàm. Còn tham số bình thường `x` dùng để chứa giá trị vào của `f` (`x` cần có giá trị trước khi `f` thực thi và sự thực thi của `f` phụ thuộc vào `x`). `x` như vậy gọi là *tham số vào*, còn `rem` gọi là *tham số ra* của hàm. Về mặt cú pháp thì trước tên tham số ra `rem` ta có thêm dấu &. Đây là đặc trưng mà C gốc không có, chỉ có trong C++ hoặc C mới[3]. Tham số có dấu & trước tên còn được gọi là *tham chiếu* còn tham số bình thường (không có dấu & trước) còn được gọi là *tham trị*. Gọi như vậy là vì khi bạn gọi hàm `f` thì đối số ứng với tham trị là biểu thức r-value, còn đối số ứng với tham chiếu phải là l-value. Đối số ứng với tham trị sẽ được lượng giá ra giá trị và được truyền (gán) cho tham số tương ứng (nên được gọi là tham trị) còn đối số ứng với tham chiếu phải là l-value, tức là một nơi chứa, để nó có thể chứa giá trị được tính và gán lúc hàm thực thi. Do đó bạn có thể gọi: `f(1.5, r)` với `r` là một biến `double`. Còn nếu gọi `f(1.5, 0.5)` thì sẽ báo lỗi do `0.5` không là l-value. Hiển nhiên bạn có thể gọi f(x, r) với `x`, `r` là các biến `double` vì `x` là một l-value thì cũng là một r-value (xem lại Bài 2.2).

Ví dụ hàm `main` sau dùng hàm `f` trên để tìm phần nguyên và phần lẻ của một số thực do người dùng nhập vào.

---

[2] Trong Toán thì được gọi là hàm đa trị.

[3] Cái gọi là C gốc, C cũ, C mới, … nghĩa là các phiên bản C, sẽ được tìm hiểu sau.

Mã 3.4.9 – Ví dụ việc dùng hàm tính phần nguyên và phần lẻ

```
int main()
{
    double x;
    printf("Nhap so thuc x: ");
    scanf("%lf", &x);

    double r;
    int d = f(x, r);
    printf("Phan nguyen la: %d. Phan le la: %lf\n",
            d, r);

    return 0;
}
```

Bạn có thấy không công bằng khi ta để phần nguyên là giá trị trả về còn phần lẻ là tham số ra (hoặc ít nhất là cũng không cân đối lắm). Hiển nhiên là bạn có thể làm ngược lại: trả về phần lẻ và để phần nguyên trong tham số ra. Cân đối hơn ta có thể để cả hai, phần nguyên và phần lẻ, là tham số ra, khi đó hàm không có giá trị trả về. Ta có thể viết lại hàm `f` như sau.

Mã 3.4.10 – Hàm tính phần nguyên và phần lẻ viết cân đối hơn:)

```
void f(double x, int &d, double &rem)
{
    d = (int)x;
    rem = x - d;
}
```

Khi đó việc gọi hàm sẽ tương tự như:

```
int d;
double r;
f(x, d, r);
```

Như vậy *giá trị trả về chính là một tham số ra*. Ta có thể dùng hàm `void` với tham số ra cho nó. Tuy nhiên ta nên xem nó là tham số ra đặc biệt. Trường hợp hàm chỉ có một tham số ra thì ta nên để nó là giá trị trả về. Trường hợp hàm có nhiều tham số ra thì ta nên để tham số ra chính (quan trọng, đặc biệt) làm giá trị trả về.

Còn một loại tham số nữa là tham số vừa vào vừa ra mà ta gọi là *tham số vào/ra*. Hàm dùng tham số vào, tính (gán) giá trị cho tham số ra. Với tham số vào/ra, hàm vừa dùng giá trị của nó vừa tính (lại giá trị mới) cho tham số đó. Chẳng hạn xét hàm g sau:

```
void g(int &d)
{
    d = d * 2;
}
```

Hàm này gấp đôi giá trị của tham số d nó nhận vào. Đây là tham số vào vì ta cần biết giá trị hiện tại để mà gấp lên, và là tham số ra vì ta đặt lại giá trị (mới) cho d. Lưu ý là về mặt cài đặt thì không có gì mới, ta dùng tham chiếu cho tham số vào/ra (cũng như đã dùng cho tham số ra). Hiển nhiên là đối số tương ứng khi gọi hàm phải là l-value. Chẳng hạn:

```
int d = 10;
g(d);
```

Sau khi chạy đoạn mã trên thì biến d chứa giá trị mới là 20. Hiển nhiên là không ai viết hàm gấp đôi giá trị như vậy vì nó quá đơn giản. Thay vì gọi `g(d)` để gấp đôi d (hiển nhiên là phải viết hàm g như trên trước) thì ta chỉ cần một dòng lệnh `d *= 2;`. Một ví dụ thực tế hơn của tham số vào ra là hàm thực hiện hoán đổi giá trị của hai tham số vào/ra.

Mã 3.4.11 – Hàm hoán đổi giá trị hai tham số

```
void swap(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
}
```

Bạn thử dùng hàm này và thử thay tham chiếu bằng tham trị (nghĩa là bỏ dấu & trước a, b) và phân tích kết quả chạy để hiểu rõ hơn nhé.

Trong Bài 3.3, điều gì xảy ra nếu ta gọi `giaidtbac2(0, 1, 1)`? Tùy môi trường, có thể bạn sẽ bị lỗi run-time hoặc có thể sẽ ra kết quả không như mong đợi. Nói chung là *không xác định* hay đúng hơn là *không có ý nghĩa*. Tại sao? Rất đơn giản, hệ số *a* phải khác 0 để nó là đa thức bậc 2. Và khi viết mã ta đã viết (hay làm) theo điều kiện này. Điều kiện này (a ≠ 0) được gọi là *giả định* của hàm `giaidtbac2`. Giả định của hàm như vậy là điều kiện ràng buộc cho các tham số của hàm để hàm có thể thực thi đúng. Ta làm sao viết giả định này? C không cung cấp cách hay để mô tả giả định của hàm. Prototype của hàm thực ra cũng đã đưa ra giả định cho các tham số và giá trị trả về (đó là nó phải có kiểu tương ứng) nhưng không đủ chi tiết như khác 0, dương, âm, … Chẳng hạn hàm `luythua` có giả định là tham số mũ phải là số nguyên không âm, trong khi prototype chỉ qui định được là kiểu `int` (số nguyên, có thể âm dương[4]). Như vậy không cấm được lời gọi hàm như `luythua(10, -4)` (mặc dù 10^{-4} là có ý nghĩa về mặt Toán, là 0.0001) nhưng ta đã không làm điều này trong hàm `luythua`, ta chỉ tính trong trường hợp n không âm.

Cách giải quyết tình huống này đó là: trong ghi chú của hàm ta nêu giả định của hàm và công bố rộng rãi trên các phương tiện thông tin đại chúng

[4] Thật ra C có kiểu số nguyên không âm mà ta sẽ tìm hiểu sau.

cho ai muốn dùng hàm này biết:) Còn lại khi ai đó dùng hàm này thì phải tuân theo giả định của hàm, tức là phải đảm bảo các đối số truyền vào phù hợp với giả định. Còn nếu vẫn cố tình gọi sai thì chương trình sẽ có lỗi run-time hoặc gì gì đó mà ta không chịu trách nhiệm. Đây là hậu quả mà người dùng hàm phải chịu do không tuân thủ qui ước của hàm. Chẳng hạn người dùng hàm `giaidtbac2` (hàm `main` ở Mã 3.3.1) đảm bảo đúng giả định này bằng cách kiểm tra $a \neq 0$ mới gọi (Dòng 9 Mã 3.3.1). Người dùng hàm `luythua` (Mã 3.4.5 và Mã 3.4.6) cũng đảm bảo giả định của hàm lũy thừa vì số mũ truyền vào (giá trị của biến lặp) là không âm do biến lặp chạy tăng từ 0. Nếu ai còn chưa rõ thì trong Toán ta có hàm exp(x) (e^x) xác định với mọi số thực x nhưng $\sqrt{x}$ chỉ xác định với số thực x không âm[5]. Như vậy x không âm là giả định của hàm $\sqrt{x}$. Chẳng hạn nếu ta viết hàm này thì sẽ như sau:

```
// Giả định: x không âm
double can(double x)
{ ... }
```

Thật sự thì hàm tính căn rất quan trọng nên đã được hỗ trợ trong thư viện chuẩn (`math.h`) với tên là `sqrt`. Bạn thử cố ý dùng hàm này sai giả định (chẳng hạn gọi `sqrt(-1)`[6]) xem thế nào nhé!

MỞ RỘNG 3.4 – Hàm có số lượng đối số tùy ý

Bạn có thể viết hàm tính tổng 2 số nguyên. Quá dễ:

```
int sum2(int a, int b)
{
    return a + b;
}
```

Khi dùng hàm này bạn truyền đúng 2 đối số (nguyên) và nhận được giá trị là tổng của 2 số đó. Chẳng hạn lời gọi `sum2(1, 2)` có giá trị 3 (là tổng của 1 và 2). Bạn có thể viết hàm tính tổng của 3 số. Cũng quá dễ:

```
int sum3(int a, int b, int c)
{
    return a + b + c;
}
```

Cũng vậy, khi dùng hàm này bạn truyền đúng 3 đối số (nguyên) và nhận

[5] Như đã nói, C có kiểu số nguyên không âm nhưng không có kiểu số thực không âm.

[6] Nếu là số phức thì $\sqrt{-1} = i$ (đơn vị ảo) nhưng trên số thực thì $\sqrt{-1}$ không xác định. Hàm `sqrt` cũng chỉ làm việc trên số thực (`double`). Hơn nữa C cũng không hỗ trợ sẵn kiểu số phức.

được giá trị là tổng của 3 số đó. Chẳng hạn lời gọi `sum3(1, 2, 3)` có giá trị 6 (là tổng của 1, 2, 3).

Thế bạn muốn viết hàm tính tổng của *n* số nguyên nói chung? Không có gì khó về mặt "thuật toán", bạn chỉ cần cộng hết *n* số đó lại. Tuy nhiên có khó khăn về mặt "kỹ thuật": bình thường, C yêu cầu các hàm phải có số lượng tham số (và do đó đối số) cụ thể và cố định. Chẳng hạn hàm `sum2` nhận đúng 2 đối số còn `sum3` nhận đúng 3 đối số (không được ít hơn hay nhiều hơn). Thậm chí `sum2` phải có tên khác `sum3` nếu chúng cùng xuất hiện trong chương trình[7]. Tuy nhiên C vẫn cho phép viết các hàm có thể nhận *số lượng đối số tùy ý*. Chẳng hạn ta viết hàm `sum` tính tổng của *n* số nguyên như sau.

Mã 3.4.12 – Hàm tính tổng *n* số nguyên dùng kĩ thuật hàm có số lượng đối số tùy ý

```
1  #include <stdarg.h>
2  
3  int sum(int n, ...)
4  {
5      va_list nums;
6      va_start(nums, n);
7  
8      int num, S = 0;
9      for(int i = 1; i <= n; i++)
10     {
11         num = va_arg(nums, int);
12         S += num;
13     }
14 
15     va_end(nums);
16 
17     return S;
18 }
```

Trong prototype của hàm `sum` (Dòng 3), sau *tham số bình thường* `n`, là dấu 3 chấm (`...`)[8]. Điều này cho biết khi gọi `sum` ta phải đưa đối số đầu tiên là số nguyên còn sau đó có thể không có, có 1, 2, … tùy ý các đối số (với kiểu tùy ý). Về ngữ nghĩa thì đối số đầu tiên là số lượng số (*n*) cần tính tổng và *n* đối số sau đó là các số cần tính tổng. Đây cũng là giả định của hàm này. Chẳng hạn ta gọi `sum(2, 1, 2)` để tính tổng 2 số 1, 2 và gọi

[7] C không cho phép các hàm trùng tên. Do đó thư viện `math.h` có hàm `fabs` để tính trị tuyệt đối của một số thực (`double`) còn hàm `abs` để tính trị tuyệt đối của một số nguyên (`int`), thậm chí là `labs` để tính trị tuyệt đối của một số nguyên `long`! Vấn đề ("nạp chồng hàm") này sẽ được tìm hiểu sau.

[8] Gõ dấu 3 chấm bằng cách gõ 3 dấu chấm liền nhau!

`sum(3, 1, 2, 3)` để tính tổng 3 số 1, 2, 3.

Trong cài đặt của `sum` thì tham số `n` sẽ nhận giá trị như thông thường và nó cho biết số lượng đối số sau đó (chính là số lượng số mà `sum` cần tính tổng). Đầu tiên `sum` cần khai báo một biến có kiểu là `va_list` để chứa các đối số thay đổi. Vì các đối số này là các số cần tính tổng nên `sum` đặt tên là `nums` như Dòng 5 cho thấy. Sau đó Dòng 6 giúp "khởi động" biến chứa các đối số thay đổi bằng macro `va_start`[9] với cú pháp như sau:

> `va_start(`<tên biến chứa các đối số thay đổi>`,` <tên tham số ngay trước dấu ba chấm>`);`

Sau đó để lấy về giá trị của từng đối số theo thứ tự trong danh sách các đối số thay đổi ta dùng macro `va_arg` với cú pháp như sau:

> `va_arg(`<tên biến chứa các đối số thay đổi>`,` <tên kiểu của đối số>`);`

Dòng 11 như vậy lấy về (lần lượt) các số (kiểu `int`) cần tính tổng trong `n` số bỏ vào biến `num` và cộng dồn vào biến `S` chứa tổng. Sau khi dùng xong các đối số thay đổi ta dùng macro `va_end` để thông báo điều đó. Kiểu `va_list` và các macro trên được khai báo trong `stdarg.h` nên ta cần `#include` nó như Dòng 1.

Cần lưu ý rằng C có các ràng buộc cho các hàm có số lượng đối số tùy ý. Trước hết các hàm này phải có ít nhất 1 tham số bình thường và dấu 3 chấm (ứng với các đối số thay đổi) phải là tham số cuối cùng. Quan trọng hơn nữa, các tham số bình thường phải giúp xác định được số lượng và kiểu của các đối số thay đổi. Với hàm `sum` thì tham số đầu tiên (cũng là tham số bình thường duy nhất) cho biết số lượng các đối số thay đổi. Hơn nữa theo giả định của hàm các đối số này đều có kiểu là `int`.

Cuối cùng bạn có tinh ý thấy rằng các hàm `printf`, `scanf` là các hàm có số lượng đối số tùy ý. Thật vậy prototype của `printf` có thể viết "nôm na" như sau[10]:

```
int printf(const char * format, ...);
```

Khi gọi `printf` ta đưa đối số đầu tiên là một chuỗi (ứng với tham số `format`) chứa thông tin định dạng. Sau đó tùy theo chuỗi định dạng mà ta sẽ đưa thêm các đối số với số lượng và kiểu phù hợp. Ở đây `printf` đã dùng các đặc tả chuyển đổi trong chuỗi định dạng (đối số đầu tiên và là đối số bắt buộc phải có) để xác định số lượng và kiểu của các đối số thay đổi sau đó (xem Bài 3.1).

---

[9] Macro là thứ ma thuật trong C mà ta sẽ tìm hiểu sau.

[10] Bạn có thể xem trong `stdio.h` tuy nhiên hơi rối.

BÀI TẬP

Bt 3.4.1 Làm lại Bài tập 3.2.1 bằng cách tổ chức hàm phù hợp.

Bt 3.4.2 Làm lại Bài tập 3.2.3 bằng cách viết riêng hàm để tính toán các giá trị và hàm `main` sẽ dùng các hàm này để xuất ra kết quả. Hơn nữa hàm `main` cũng xuất ra các giá trị cho trước trong ghi chú (như Bài tập 3.2.3(c) là sin$x$) để đối chiếu kết quả.

Bt 3.4.3 Viết hàm xác định đồng thời số nhỏ và lớn trong hai số được cho theo hai cách:

- 2 số được cho để trong 2 tham số vào và số nhỏ, lớn để trong 2 tham số ra.
- 2 tham số vào/ra chứa 2 số được cho trước khi thực thi hàm và chứa số nhỏ, số lớn sau khi thực thi hàm.

Dùng hàm này để tìm số nhỏ nhất và lớn nhất trong *n* số. So sánh ưu khuyến điểm của hai cách làm.

Bt 3.4.4 Viết lại hàm `sum` trong Mở rộng 3.4 (Mã 3.4.12) để tính tổng các số thực (`double`).

Bt 3.4.5 Tương tự hàm `sum` trong Mở rộng 3.4 (Mã 3.4.12) viết hàm `max` tìm số lớn nhất trong các số nguyên. Chẳng hạn `max(3, 1, 2, 3)` trả về 3.

Bt 3.4.6 Tương tự hàm `sum` trong Mở rộng 3.4 (Mã 3.4.12) viết hàm `printvector` xuất ra một vector các số nguyên. Chẳng hạn `printvector (3, 1, 2, 3)` xuất ra `(1, 2, 3)`.

Bt 3.4.7 Viết lại hàm `sum` trong Mở rộng 3.4 (Mã 3.4.12) bằng cách dùng số 0 đánh dấu kết thúc các số (chứ không dùng số lượng số như đã làm). Chẳng hạn để tính tổng các số 1, 2, 3 ta gọi `sum(1, 2, 3, 0)` (chứ không phải `sum(3, 1, 2, 3)` như đã làm).

BÀI 3.5

Ta đã thấy mô hình cơ bản của chương trình: nhập dữ liệu, xử lý dữ liệu, xuất dữ liệu. Ta có thể xem chương trình như là một *máy chế biến dữ liệu*. Nếu dữ liệu là nguyên liệu (gạo, tôm, cá, rau, …) thì chương trình là người đầu bếp[1]. Có hai thứ nữa liên quan đến *dữ liệu* là *thông tin* và *tri thức*. Và thường thì người ta phân biệt dữ liệu với thông tin với tri thức. Chẳng hạn một chương trình nhận dữ liệu là điểm thi của các thí sinh cho biết thông tin là thí sinh cao điểm nhất và cho biết tri thức là "thí sinh vùng khó khăn thường học giỏi hơn thí sinh ở thành phố". Tuy nhiên ranh giới giữa dữ liệu/thông tinh/tri thức là không rõ ràng và phụ thuộc ngữ cảnh cũng như quan điểm, góc nhìn. Trong tài liệu này, mọi thứ đều là dữ liệu và như vậy chương trình là văn bản mô tả cách thức tổ chức và xử lý dữ liệu trên máy.

Việc đầu tiên khi làm việc với dữ liệu là *phân loại hay phân dạng dữ liệu*. Ta có những loại hay dạng dữ liệu nào? Chương trình tính toán như Calculator thì xử lý dữ liệu số, chương trình như Word thì xử lý dữ liệu văn bản, các chương trình giải trí/multimedia thì xử lý nhiều loại dữ liệu kết hợp như hình ảnh, âm thanh, … Dữ liệu xử lý trên máy rõ ràng rất đa dạng và thường pha trộn lẫn nhau như một văn bản trong Word ngoài các kí tự (phần văn bản thô) còn có dữ liệu định dạng (màu sắc, font chữ, …) hay các hình ảnh được chèn, … Một nhóm các dữ liệu giống nhau về ý nghĩa, mục đích sử dụng, dạng thức thì được gọi là một *kiểu dữ liệu*. *Các dữ liệu của cùng một kiểu sẽ được tổ chức và xử lý giống nhau trên máy*. Chẳng hạn -2, -1, 0, 1, 2, … là các dữ liệu cụ thể khác nhau nhưng chúng đều là các số nguyên, chúng được tổ chức và xử lý giống nhau trên máy. Ta nói rằng chúng có kiểu dữ liệu là kiểu số nguyên (có dấu, tức là có số âm). Một kiểu dữ liệu như vậy xác định: các dữ liệu cụ thể có thể có của kiểu này, tập các thao tác xử lý có thể dùng trên dữ liệu của kiểu, cách định dạng và lưu trữ/truyền nhận dữ liệu của kiểu trên máy. Một kiểu dữ liệu cũng thường được định danh bằng một tên gọi là *tên kiểu*. Rõ ràng ta có rất nhiều kiểu dữ liệu được phân loại theo nhiều tiêu chí khác nhau: đơn giản/phức tạp, cụ thể/trừu tượng, có sẵn/tự định nghĩa, … Trong bài này ta tìm hiểu những kiểu dữ liệu đơn giản nhất, *cơ bản* nhất và được *hỗ trợ sẵn* trong C.

Kiểu dữ liệu nào là cơ bản? Ở mức thấp nhất, trên một máy số thì mọi dữ liệu đều là dãy *bít*. Tuy nhiên các bít không được xử lý riêng lẻ mà

[1] Đúng hơn thì chương trình là công thức nấu ăn còn máy thực thi chương trình là người đầu bếp nấu theo công thức.

thường được gom lại thành các đơn vị lớn hơn như *byte* (8 bít), *word* (một vài byte: 2/4/8 byte), *double word* (2 word). Các word được xem là đơn vị dữ liệu của máy (gọi là *từ máy*) và nó được xem như là các số nguyên (có dấu và không dấu). Nói chung với đa số *kiến trúc máy* thì số nguyên (và số thực) là dữ liệu cơ bản (với các chỉ lệnh được hỗ trợ để thao tác trên số nguyên/số thực như cộng trừ nhân chia…). Như vậy kiểu dữ liệu cơ bản là số nguyên/số thực. Điều này cũng phản ánh qua lịch sử phát triển của máy tính, nó là một máy tính (máy tính toán (thao tác) trên các con số). Kiểu dữ liệu cơ bản kế tiếp là kiểu luận lý. Kiểu này chỉ có hai dữ liệu là Đúng/Sai. Trên máy không trực tiếp có kiểu này nhưng các chỉ lệnh nhảy đều dựa trên một điều kiện nào đó mà việc thực thi tùy thuộc vào điều kiện đó là đúng hay sai. Hơn nữa trong quá trình phát triển thì văn bản cũng là dạng thức dữ liệu phổ biến trên máy. Đơn vị của văn bản chính là kí tự. Và đây cũng là kiểu dữ liệu cơ bản. Tóm lại có 4 kiểu dữ liệu được xem là cơ bản[2]: số nguyên, số thực, luận lý, kí tự[3]. Ta còn gọi một dữ liệu cụ thể của kiểu cơ bản là *giá trị*.

C là ngôn ngữ rất gần gũi với kiến trúc máy. Điều này phản ánh qua sự hỗ trợ các kiểu dữ liệu cơ bản của nó: các kiểu dữ liệu cơ bản trên đều là kiểu có sẵn (built-in) trong C và được tổ chức/xử lý trực tiếp trên kiến trúc máy. Nghĩa là định dạng giống trên máy và các thao tác hầu như được biên dịch trực tiếp thành các chỉ lệnh tương ứng. Có thể nói rằng trên C (cũng như trên máy) chỉ có số (nguyên và thực) mà thôi. Mọi dữ liệu khác (kể cả luận lý, kí tự) đều được xây dựng dựa trên số.

Số nguyên là số nguyên, tức không có phần lẻ như số thực. Các số nguyên không âm, mà ta còn gọi là số tự nhiên, là các số cơ bản nhất. Chúng còn được gọi là số đếm vì chúng được dùng chủ yếu để đếm. Việc dùng thêm các số âm cho phép ta biểu diễn rất nhiều đại lượng như số tiền (dương là có còn âm là nợ), … Trên số nguyên có các thao tác (phép toán) quen thuộc như cộng, trừ, nhân, chia. Riêng phép chia thì là chia nguyên: chia được thương (nguyên) và số dư. Mọi dữ liệu tổ chức trên máy đều được biểu diễn dưới dạng dãy bít. Hiển nhiên là dãy bít này là hữu hạn khi lưu trữ trên vùng nhớ. Với các dữ liệu cơ bản ta sẽ thấy rằng dãy bít biểu diễn chúng không chỉ hữu hạn mà còn cố định và thường khá ngắn (để thao tác dễ dàng và tiết kiệm bộ nhớ cũng như kênh truyền dẫn). Như vậy trên C có các kiểu số nguyên khác nhau tùy theo chúng có số âm hay không (gọi là có dấu) và kích thước của chúng (số bít dùng để biểu diễn) như bảng sau đây cho thấy.

Bảng 3.5.1 – Bảng liệt kê các kiểu số nguyên trong C

[2] Đây là theo ý tôi, tùy người/góc nhìn/ngữ cảnh mà có ý kiến khác nhau.
[3] Còn một kiểu dữ liệu nữa cũng rất cơ bản và trực tiếp trên máy là địa chỉ mà ta sẽ tìm hiểu sau.

Tên kiểu[4]	Kích thước	Giá trị nhỏ nhất	Giá trị lớn nhất
`(signed) char`	8 bít (1 byte)	-128 -2^7	127 $2^7 - 1$
`unsigned char`		0	255 $2^8 - 1$
`(signed) short`	16 bít (2 bye)	-32 768 -2^{15}	32 767 $2^{15} - 1$
`unsigned short`		0	65 535 $2^{16} - 1$
`(signed) int`	32 bít (4 byte)	-2 147 483 648 -2^{31}	2 147 483 647 $2^{31} - 1$
`unsigned int`		0	4 294 967 295 $2^{32} - 1$
`(signed) long`	64 bít (8 byte)	(xem chú thích)[5] -2^{63}	(xem chú thích)[6] $2^{63} - 1$
`unsigned long`		0	(xem chú thích)[7] $2^{64} - 1$

Giá trị lớn nhất và nhỏ nhất của một kiểu số nguyên xác định *phạm vi* của kiểu số nguyên đó. Giá trị (dữ liệu) của kiểu phải nằm trong phạm vi này, nghĩa là không được nhỏ hơn giá trị nhỏ nhất hay lớn hơn giá trị lớn nhất. Chẳng hạn kiểu `unsigned char` có phạm vi từ 0 đến 255 nên các giá trị có thể của kiểu là {0, 1, 2, ..., 254, 255}. Các số nguyên âm không thuộc kiểu này cho nên kiểu này gọi là *kiểu số nguyên không dấu*. Tương tự ta có các kiểu số nguyên không dấu khác là `unsigned short/int/long` đều có giá trị từ 0 trở lên. Tại sao giá trị lớn nhất của kiểu `unsigned char` là 255? Ta chưa đi sâu vào biểu diễn cụ thể trên máy của các kiểu này[8]. Tuy nhiên rất dễ hiểu thôi: kiểu này có kích thước là 8 bít (tức là 1 byte) tức là máy sẽ dùng 8 bít để biểu diễn một giá trị của kiểu này. Mỗi bít có thể ở một trong 2 trạng thái khác nhau là 0/1 (hay cao/thấp) nên 8 bít có thể ở $2^8 = 256$ trạng thái khác nhau. Mỗi trạng thái khác nhau này sẽ biểu diễn cho một số nguyên khác nhau nên 256 trạng thái khác nhau sẽ biểu diễn được cho 256 số nguyên khác nhau là 0, 1, ..., 255. Kiểu `signed char` cũng có kích thước là 1 byte nhưng được dùng để biểu diễn cho 256 số nguyên là {-128, -127, ..., -1, 0, 1, ..., 126, 127}. Một nửa số trạng thái ($2^8/2= 2^7$) dùng để biểu diễn cho số âm

[4] Cũng đồng thời là từ khóa trong C.
[5] -2^{63} = -9 223 372 036 854 775 808.
[6] $2^{63} - 1$ = 9 223 372 036 854 775 807.
[7] $2^{64} - 1$ = 18 446 744 073 709 551 616.
[8] Ta sẽ và phải tìm hiểu kĩ ở tầng cao hơn.

nên số âm nhỏ nhất là -2^7. Một nửa số trạng thái dùng để biễu diễn cho số không âm nên giá trị lớn nhất là $2^7 - 1$ (số 0 cũng là số không âm). Vì kiểu `signed char` cho phép giá trị âm (≥ -128) nên được gọi là *kiểu có dấu* (signed). C dùng các từ khóa tương ứng cho các kiểu trên, nên để khai báo một biến số nguyên có thể chứa các số nguyên nhỏ nhỏ (có thể âm), chẳng hạn từ -100 đến 100 thì ta có thể dùng kiểu `signed char` và khai báo như sau:

```
signed char a;
```

Hơn nữa C mặc định kiểu số nguyên là có dấu nên khai báo trên có thể viết ngắn gọn là: `char a;`. Để khai báo kiểu không dấu thì ta phải viết rõ từ khóa `unsigned` đằng trước như: `unsigned char a;`

Những thảo luận trên cho kiểu số nguyên 1 byte có dấu và không dấu (`char`/`unsigned char`) cũng áp dụng cho các kiểu số nguyên với kích thước khác: `short` (2 byte), `int` (4 byte), `long` (8 byte). Chẳng hạn bạn chắc hiểu được tại sao giá trị nhỏ nhất của kiểu `int` (có dấu, `signed int`) là -2 147 483 648.

Ta cũng phải lưu ý là lựa chọn kích thước cho các kiểu số nguyên trên còn tùy thuộc kiến trúc máy và compiler. Chẳng hạn trên máy 16-bít thì thường kiểu `int` là 2 byte còn kiểu `long` là 4 byte, trên máy 32-bít thì kiểu `int` là 4 byte và kiểu `long` cũng 4 byte, trên máy 64-bít thì kiểu `int` là 4 byte còn kiểu `long` là 8 byte. Như vậy bảng trên là trường hợp điển hình cho máy 64-bít. Để kiểm tra cụ thể kích thước của từng kiểu thì các bạn dùng toán tử `sizeof`, cú pháp: `sizeof(<tên kiểu>)` hoặc `sizeof(<tên biến>)` sẽ trả về kích thước tính theo byte của kiểu tương ứng với tên kiểu hoặc kiểu ứng với kiểu được khai báo của biến. Chẳng hạn bạn có thể chạy thử dòng lệnh sau:

```
printf("%d", sizeof(long));
```

để biết kích thước cụ thể của kiểu `long` trên máy nào đó. File `limits.h` cũng cho biết phạm vi cụ thể mà compiler chọn cho các kiểu này. Chẳng hạn hằng tượng trưng `INT_MIN` được `#define` trong `limits.h` cho giá trị nhỏ nhất kiểu `int` (mà nếu trong trường hợp điển hình trên chính là -2,147,483,648 (-2^{31}), bạn cũng thử xuất ra giá trị này trên máy mình nhé). Cũng lưu ý một số compiler C mới[9] có thêm kiểu `long long` (có dấu và không có dấu) mà trên máy 32-bít thường là 8 byte.

Nói chung nhiều kiểu số nguyên mang lại nhiều lựa chọn cho ta. Chẳng hạn nếu ta chỉ làm việc với số không âm thì nên dùng kiểu không dấu (giúp phạm vi rộng thêm). Ta cũng nên dùng kiểu có kích thước vừa đủ rộng để tiết kiệm bộ nhớ. Với trường hợp thông thường ta nên dùng kiểu `int` và khi cần số lớn hơn thì có thể dùng kiểu `long long`. Vậy trường hợp ta cần số

[9] Compiler hỗ trợ C99 mà ta sẽ bàn sau.

siêu khủng thì sao? Một số compiler C mới có cung cấp kiểu số nguyên 128 bít mà ta có thể tận dụng hoặc là ta sẽ tự mình định nghĩa (tạo ra) kiểu này. Ta sẽ bàn thêm điều này sau.

Các thao tác (phép toán) cơ bản trên số nguyên được C cung cấp bằng các toán tử mà bạn đã biết trong Bài 2.1, Bài 2.2 và một số toán tử khác mà bạn sẽ biết sau. Trường hợp bạn cần thực hiện một thao tác nào đó mà chưa được C cung cấp thì sao? Bạn sẽ tự mình viết lấy bằng cách viết hàm cho thao tác đó. Chẳng hạn ta viết hàm `abs` sau để cung cấp thao tác tính trị tuyệt đối của một số nguyên[10].

Mã 3.5.1 – Hàm tính trị tuyệt đối của một số nguyên

```
int abs(int x)
{
    if(x < 0)
        return -x;
    return x;
}
```

Sau đó để tính trị tuyệt đối của một số nguyên nào đó ta sẽ dùng hàm này thay vì dùng toán tử như các thao tác cộng, trừ, ….

Bạn đã biết cách mô tả một hằng số nguyên trong Bài 2.1. Bạn cũng đã biết các hỗ trợ nhập xuất cho số nguyên trong Bài 3.1. Bạn cũng đã biết vấn đề tràn số nguyên (Mở rộng 1.2). Nói chung bạn phải đảm bảo rằng ta làm việc với giá trị trong phạm vi của kiểu để kết quả tin cậy. Nếu không ta cần dùng kiểu có kích thước lớn hơn. Chẳng hạn ta nên dùng kiểu `long long` để chứa giai thừa của một số vì giá trị này thường rất lớn như đoạn mã sau minh họa[11].

Mã 3.5.2 – Đoạn mã tính giai thừa của một số nguyên với kết quả rất lớn

```
int n = 20;
long long f = 1;
for(int i = 1; i <= n; i++)
    f *= i;
printf("%lld", f);
```

Kết quả xuất ra giá trị của 20! là: 2432902008176640000. Một con số rất lớn. Đoạn mã trên không có gì nhiều ngoài chuyện ta đã dùng kiểu `long long` (mà kích thước là 64-bít cho phạm vi rất lớn) và dùng định dạng xuất `"%lld"` cho kiểu này[12]. Tuy nhiên 20! cũng là "kịch trần" của kiểu này rồi. Bạn thử với 21! sẽ rõ[13]. Vậy để tính cỡ 100! thì làm sao ta?

---

[10] Thư viện chuẩn của C (`math.h`) có hàm `fabs` để tính trị tuyệt đối nhưng là của số thực `double`.

[11] Compiler của bạn phải hỗ trợ `long long` với kích thước 64 bít.

[12] Ví dụ này cũng cho thấy hàm giai thừa lớn nhanh cỡ nào.

[13] Ra kết quả không đúng (vì tràn số) đó.

Kiểu dữ liệu khác cũng quan trọng không kém số nguyên, nhất là trong tính toán khoa học, đó là *kiểu số thực*. Số thực cho phép mô tả các số mà có thể không nguyên. Chẳng hạn số 1.5 có phần lẻ là 0.5 (còn phần nguyên là 1). Số thực dùng cho hầu hết các đại lượng, kết quả tính toán trong thực tế, kỹ thuật và khoa học. Trước hết, số thực trên máy cũng hữu hạn như số nguyên, chẳng hạn số 1/3 không thể biểu diễn chính xác bằng số thực trên máy được do số này có khai triển thập phân là vô hạn (1/3 = 0.3333333…). Thật ra thì con số đơn giản như 0.1 cũng không thể biểu diễn trên máy vì nó cũng được khai triển nhị phân là vô hạn[14]. Hơn nữa vì số trạng thái dùng để biểu diễn cũng hữu hạn nên số lượng số thực cũng hữu hạn do đó có nhiều số thực sẽ được biểu diễn gần đúng. Để hiểu điều này ta sẽ phải tìm hiểu cách chuẩn hóa một số thực. Cho số thực x khác 0 ta chuẩn hóa bằng cách viết x thành $(-1)^s \times y \times 10^e$. Trong đó y là số thực sao cho $1 \leq y < 10$. Khi đó s được gọi là phần dấu: s = 1 cho số âm và s = 0 cho số dương, e được gọi là phần mũ, y được gọi là phần trị. Các chữ số của y không kể các số 0 bên trái và bên phải là các *chữ số có nghĩa*. Khi biểu diễn trên máy thì các phần dấu, mũ và trị được biểu diễn riêng biệt[15]. Chẳng hạn số a = -0.000001 sẽ được viết thành dạng chuẩn là $(-1)^1 \times 1 \times 10^{-6}$ có phần dấu là 1 (âm), phần mũ là -6 và phần trị là 1. *Số chữ số có nghĩa* là 1 chữ số (1 có thể viết là 001.00 nhưng ta bỏ các số 0 bên trái và bên phải). Số b = 1010000 sẽ được viết thành dạng chuẩn là $(-1)^0 \times 1.01 \times 10^6$ có phần dấu là 0 (dương), phần mũ là 6 và phần trị là 1.01. Số chữ số có nghĩa là 3 chữ số. Số $c = b - a$ = 1010000.000001 sẽ được viết thành dạng chuẩn là $(-1)^0 \times 1.010000000001 \times 10^6$ có phần dấu là 0 (dương), phần mũ là 6 và phần trị là 1.010000000001. Số chữ số có nghĩa là 13 chữ số. Vì phần trị được tách riêng và cũng hữu hạn (và chỉ được phép có rất ít chữ số) nên có thể khi lưu trữ sẽ cắt bỏ bớt các chữ số ý nghĩa bên phải. Chẳng hạn phần trị của c có thể chỉ còn là 1.01 do cắt bỏ chữ số có nghĩa tận cùng bên phải.

Trên máy (và trong C) ta không có kiểu số thực không âm, tức là chỉ có kiểu số thực có dấu. Nhưng tùy theo kích thước (số bít lưu trữ) của các phần mũ và trị mà ta có các kiểu số thực khác nhau như bảng sau.

Bảng 3.5.2 – Bảng liệt kê các kiểu số thực trong C

Kiểu	Kích thước	Số chữ số có nghĩa tối đa	Số mũ nhỏ nhất	Số mũ lớn nhất
`float`	4 byte	6 chữ số	-38	38
`double`	8 byte	15 chữ số	-308	308

[14] Số 0.1 (cơ số 10) biểu diễn nhị phân (cơ số 2) là: 0.0001100110011… vô hạn tuần hoàn (0011 lặp lại), ta sẽ hiểu hơn sau.

[15] Ở đây để dễ hiểu tôi đã làm theo cơ số 10 nhưng trên máy sẽ là cơ số 2, ta sẽ trở lại vấn đề này sau.

Số chữ số có nghĩa tối đa còn được gọi là *độ chính xác* của kiểu số thực vì khi phần trị có nhiều chữ số có nghĩa thì sẽ chỉ giữ lại các chữ số bên trái và cắt bỏ các chữ số bên phải. Chẳng hạn phần trị 1.010000000001 sẽ được cắt bỏ còn 1.01 trong kiểu `float` (do kiểu `float` chỉ cho phép tối đa 6 chữ số có nghĩa) nhưng vẫn còn giữ đầy đủ trong kiểu `double` (do kiểu `double` cho phép tới 15 chữ số có nghĩa). Các bạn chạy đoạn mã sau và suy ngẫm để hiểu rõ nhé.

Mã 3.5.3 – Đoạn mã minh họa độ chính xác của kiểu số thực

```
float a = -0.000001;
float b = 1010000;
float c = b - a;
double d = b - a;
printf("%g %g %.15g %.15g", a, b, c, d);
```

Bạn cũng đã biết cách viết hằng số thực cũng như nhập/xuất số thực. Cũng lưu ý là có compiler cung cấp kiểu `long double` với kích thước 10 byte cho độ chính xác lớn hơn và phạm vi số mũ cũng lớn hơn nhưng có thể kiểu `long double` cũng là kiểu `double`. Tùy compiler à! Bạn có thể tham khảo file `float.h` để xem các lựa chọn cụ thể.

Một cách để tính 100! đó là ta sẽ dùng kiểu `double` như đoạn mã sau.

Mã 3.5.4 – Đoạn mã tính giai thừa của một số nguyên với kết quả siêu lớn

```
int n = 100;
double f = 1;
for(int i = 1; i <= n; i++)
    f *= i;
printf("%.15g", f);
```

Kết quả là: 100! = 9.33262154439441e+157 (nghĩa là $9.33262154439441 \times 10^{157}$). Dĩ nhiên số chữ số có nghĩa của 100! sẽ rất nhiều vượt quá độ chính xác 15 chữ số của kiểu `double`. Tuy nhiên phần mũ 157 vẫn trong phạm vi mũ của kiểu `double` (-308 tới 308). Kết quả trên cho ta biết 100! là số nguyên viết theo cơ số 10 có 158 chữ số trong đó 15 chữ số hàng cao nhất (quan trọng nhất) là 933262154439441 (lưu ý các thông tin này đều chính xác, ta chỉ không biết các chữ số có nghĩa còn lại sau đó). Vậy nếu bạn muốn biết chính xác 158 chữ số của 100! thì sao? Ta sẽ giải quyết ở tầng cao hơn.

Kiểu dữ liệu cơ bản tiếp theo là *kiểu luận lý*. Kiểu này thật ra là kiểu đơn giản nhất. Nó chỉ có hai giá trị (dữ liệu) là {Đúng, Sai}[16]. Các giá trị của kiểu luận lý còn gọi là *chân trị* vì nó dùng để mô tả Đúng hoặc Sai (True hoặc False). Tuy nhiên nó cũng có thể dùng cho mục đích khác như Có hoặc Không (Yes hoặc No) hay Cao hoặc Thấp (High hoặc Low), …. Đơn giản nhất

[16] Trong khi kiểu số nguyên đơn giản nhất là kiểu `unsigned char` có tới 256 giá trị khác nhau {0, 1, …, 255}.

ta kí hiệu nó là 1 hoặc 0. Kiểu luận lý được dùng cho các điều kiện, các yêu cầu, các kiểm tra, … mà kết quả là Đúng hoặc Sai. C không có kiểu riêng cho kiểu luận lý (không có từ khóa nào cho kiểu này) tuy nhiên C dùng kiểu số (số nguyên/thực) cho kiểu luận lý với qui ước: số 0 (hay 0.0) là sai và khác 0 (hay khác 0.0) là đúng. C cũng có toán tử cho kết quả luận lý như các toán tử so sánh mà giá trị là đúng (số nguyên 1) hay sai (số nguyên 0). Các thao tác trên kiểu luận lý cũng đơn giản, quen thuộc trong Logic và được C hỗ trợ bằng các toán tử luận lý tương ứng: và (&&), hoặc (||), phủ định (!) (xem lại Bài 2.1).

Trong C kiểu luận lý xuất hiện trong các điều kiện của các lệnh lựa chọn hay lệnh lặp. Nó cũng có thể được dùng trực tiếp như các dữ liệu khác trong các biến "*cờ*" (tức là các biến cho biết một sự kiện nào đó đã xảy ra hay chưa) hay giá trị luận lý trả về của một hàm. Chẳng hạn hàm sau:

Mã 3.5.5 – Hàm kiểm tra tính nguyên tố

```
int isprime(int n)
{
    for(int i = 2; i < n; i++)
        if(n % i == 0)
            return 0;
    return 1;
}
```

được khai báo kiểu trả về là `int` và nó trả về giá trị nguyên 0/1 nhưng ta phải hiểu là nó trả về giá trị luận lý Sai/Đúng. Hàm này nhận một số nguyên n (với giả định n ≥ 2) và trả về Đúng (1) nếu n là số nguyên tố và ngược lại, trả về Sai (0) nếu n không là số nguyên tố (tức là hợp số). Như vậy nó là *hàm kiểm tra* tính nguyên tố[17].

Kiểu quan trọng cuối cùng mà ta cũng xem là cơ bản là *kiểu kí tự*. Đây là kiểu cơ bản để xây dựng nên kiểu quan trọng là *chuỗi*. Mà chuỗi (hay văn bản) là dạng dữ liệu quen thuộc nhất với ta (có lẽ hơn cả số) và cũng phổ biến nhất trên máy. Tuy nhiên C cũng không có kiểu riêng cho kiểu kí tự. C dùng số nguyên để biểu diễn kí tự. Cụ thể là dùng mã số của kí tự. Mỗi kí tự sẽ được gắn với một con số qui ước, gọi là *mã kí tự* (tương tự như ta dùng số chứng minh thư cho công dân hay mã số sinh viên cho sinh viên). *Bảng mã ASCII* là bảng mã phổ biến cho qui ước mã số này (xem lại Bài 1.7). Bảng mã ASCII có nhược điểm là chỉ mã cho 128 kí tự (đây là vấn đề lịch sử do các kí tự ở nước Âu, Mỹ rất ít). Trong thời đại toàn cầu ngày nay thì có thể nói bộ ký tự là của cả hành tinh và như vậy thì nó có rất nhiều kí tự. Bãng mã ASCII này đã được mở rộng ra thành *bảng mã Unicode* mà ta sẽ tìm hiểu sau. C dùng số nguyên 1 byte (kiểu `char` hoặc `unsigned char`) để quản lý

[17] Các hàm kiểm tra thường được đặt tên với tiếp đầu ngữ là `is_`. Những hàm trả về giá trị luận lý được gọi chung là hàm quyết định.

mã của kí tự. Cho nên kiểu kí tự trong C thật ra là kiểu số nguyên (1 byte) nhưng vì kiểu số nguyên 1 byte dùng chủ yếu cho mục đích này[18] nên C đặt tên nó là kiểu `char` luôn[19].

Bạn cũng đã biết cách mô tả một hằng kí tự và nhập/xuất kí tự trong Bài 3.1. Các thao tác trên kí tự được cài đặt chủ yếu dựa trên việc bố trí bảng mã ASCII chẳng hạn thao tác tìm kí tự chữ cái hoa tương ứng với kí tự chữ cái thường (chữ cái hoa của chữ cái *a* là chữ cái *A*) có thể được viết như hàm sau.

Mã 3.5.6 – Hàm tìm chữ cái hoa

```
char toupper(char c)
{
    if('a' <= c && c <= 'z')
        return c + ('A' - 'a');
    return c;
}
```

Chẳng hạn lời gọi `toupper('a')` sẽ trả về `'A'` (mã ASCII của kí tự *A*, 65) còn lời gọi `toupper('1')` sẽ trả về `'1'` (mã ASCII của kí tự 1) do đây không phải là kí tự chữ cái thường. Thật sự thì thư viện chuẩn có sẵn hàm này trong `ctype.h`.

Các kiểu dữ liệu số (số nguyên, số thực và kể cả luận lý, kí tự vì thật ra trong C chúng là số) được C gọi chung là kiểu số hay *kiểu số học*.

C là ngôn ngữ *định kiểu mạnh* và *tĩnh*: tất cả dữ liệu đều phải xác định kiểu lúc biên dịch. Điều này thể hiện ở việc tất cả các biến phải được khai báo kiểu, tất cả các tham số của hàm và giá trị trả về của hàm phải được xác định kiểu qua prototype của hàm. Các hằng số cũng có kiểu: các hằng số nguyên có kiểu `int` còn các hằng số thực có kiểu là `double`[20]. Hơn nữa mọi biểu thức của C cũng đều có kiểu. *Kiểu của biểu thức* được suy từ kiểu của các toán hạng và ngữ nghĩa của phép toán tương ứng. Chẳng hạn với a là biến kiểu `double` thì biểu thức `a * 1.0` sẽ có kiểu double do a, `1.0` có kiểu `double` và do ngữ nghĩa của phép nhân: nhân hai `double` được giá trị `double`. Biểu thức `a < 1.0` có kiểu là kiểu số nguyên do biểu thức này có giá trị là đúng (1) hoặc sai (0). Tuy nhiên là kiểu số nguyên cụ thể nào thì tùy compiler chọn và thông thường là 1 byte (kiểu `char`) hay 4 byte kiểu `int`. Không ngạc nhiên khi ta có thể dùng toán tử `sizeof` với cú pháp: sizeof(<biểu thức>) để xác định kích thước của giá trị của biểu thức, cũng

[18] Hiển nhiên là bạn có thể dùng nó cho mục đích khác nhưng nhớ là nó có phạm vi rất hẹp.

[19] Viết tắt của character.

[20] Trừ khi bạn xác định khác đi như các hằng số thực có kí hiệu `f` ở cuối sẽ có kiểu là `float` hay các hằng số nguyên có kí hiệu `l` ở cuối sẽ có kiểu là `long`.

là kích thước của kiểu của biểu thức[21]. Ta có thể chạy thử đoạn mã sau để biết:

```
double a = 1.0;
printf("%d %d\n", sizeof(a * 1.5), sizeof(a < 1.5));
```

Bạn cũng đã biết rằng có biểu thức rất đặc biệt là biểu thức không có giá trị. Đó là biểu thức có kiểu void. Đây là kiểu rất đặc biệt mà ta sẽ tìm hiểu thêm sau, hiện giờ có thể nói rằng đó là một kiểu hình thức, không có dữ liệu nào thuộc kiểu này, không có vùng nhớ nào được cấp phát cho dữ liệu nào của kiểu (vì không có dữ liệu nào). Cho nên ta không thể dùng sizeof cho kiểu này hay cho biểu thức kiểu này cũng như không thể dùng các toán tử nào khác cho kiểu này (không thể thao tác trên dữ liệu kiểu này). Kiểu này chủ yếu dùng cho giá trị trả về của hàm để nói rằng hàm không trả về giá trị và khi đó hàm này có ý nghĩa qua hiệu ứng lề của nó, và ta đã gọi hàm như vậy là thủ tục.

Các bạn đã thấy các kiểu dữ liệu cơ bản rất gần gũi nhau và chúng có thể được chuyển đổi qua lại. Chẳng hạn trong biểu thức a/2 nếu a là biến kiểu int thì phép chia trên là chia nguyên do cả 2 toán hạng đều có int (2 là hằng số kiểu int). Chẳng hạn a đang chứa giá trị 5 thì kết quả là số nguyên 2. Tuy nhiên nếu a là biến kiểu double thì C không thể trực tiếp thực hiện phép chia trên do không có phép chia số kiểu double cho số kiểu int. C sẽ tự động chuyển giá trị 2 từ kiểu int thành giá trị 2.0 kiểu double và thực hiện phép chia số double cho số double. Chẳng hạn nếu a đang chứa giá trị 5.0 thì kết quả là số double 2.5. Tương tự kết quả của lệnh gán: int a = 2.5; là a chứa giá trị 2 nguyên. Giá trị 2.5 có kiểu double và không thể bỏ vào vùng nhớ của biến a kiểu int[22]. Do đó C cũng sẽ tự động chuyển giá trị 2.5 kiểu double thành giá trị 2 kiểu int (chuyển kiểu ở đây là cắt bỏ phần lẻ) rồi mới gán giá trị 2 (kiểu int) vào biến a kiểu int.

Nói chung *chuyển kiểu* là việc chuyển một giá trị (dữ liệu) từ kiểu này (mà ta gọi là kiểu nguồn) sang một kiểu khác (mà ta gọi là kiểu đích). Nó có thể là đơn giản hoặc phức tạp (như chuyển giá trị double 2.5 sang giá trị int 2 vì cần phải định dạng lại) và xảy ra khi C cần dữ liệu kiểu này nhưng lại nhận được dữ liệu kiểu khác. Cụ thể chuyển kiểu có thể xảy ra ở 4 tình huống:

(1) Khi các toán hạng của phép toán khác kiểu hay một phép toán đòi hỏi kiểu này nhưng lại được nhận kiểu khác. Ví dụ đầu tiên của ta ở trên là một trường hợp như vậy.

---

[21] Rất tiếc C không có toán tử để lấy thông tin kiểu như typeof của Java, nhưng nhớ rằng mọi biểu thức trong C đều có kiểu.

[22] Vùng nhớ của biến kiểu int có kích thước 4 byte nhưng giá trị double có kích thước 8 byte hơn nữa định dạng 2 kiểu rất khác nhau.

(2) Khi kiểu của biểu thức bên phải phép gán không khớp với kiểu của l-value bên trái phép gán. Ví dụ phép gán trên của ta là một trường hợp.
(3) Khi kiểu của các đối số (là các biểu thức) trong lời gọi hàm không khớp với kiểu của các tham số khai báo trong prototype của hàm.
(4) Khi kiểu của biểu thức trả về (trong lệnh `return`) không khớp với kiểu trả về trong prototype của hàm.

Hai trường hợp cuối xảy ra như trong ví dụ sau. Chẳng hạn có hàm f như sau:

```
int f(int v)
{
    return v + 0.5;
}
```

thì với lời gọi hàm `f(1.4)` sẽ có 3 lần chuyển kiểu xảy ra. Đầu tiên đối số `1.4` có kiểu `double` không khớp với kiểu của tham số của hàm `f` (kiểu `int`). Do đó giá trị 1.4 kiểu `double` sẽ được chuyển thành giá trị 1 kiểu `int` và như vậy v sẽ có giá trị 1 trước khi hàm `f` thực thi (trường hợp chuyển kiểu thứ 3 ở trên). Khi `f` thực thi thì ở biểu thức `v + 0.5`, giá trị của v (1) sẽ được chuyển thành kiểu `double` (1.0) để cộng với 0.5 (do `0.5` có kiểu `double`) (trường hợp thứ 1 ở trên). Sau đó giá trị của biểu thức `v + 0.5` là 1.5 kiểu `double` sẽ là giá trị trả về của hàm nhưng hàm lại được khai báo kiểu trả về là `int` nên giá trị này sẽ được chuyển thành giá trị 1 kiểu `int` (trường hợp thứ 4 ở trên). Như vậy giá trị của `f(1.4)` là 1 kiểu `int`.

Việc chuyển kiểu ta gặp qua các ví dụ trên được gọi là *chuyển kiểu tự động* hay *ngầm định* do C tự thực hiện. Ta (lập trình viên) cũng có thể yêu cầu (bắt buộc) C chuyển kiểu, gọi là *chuyển kiểu tường minh* bằng *toán tử ép kiểu* với cú pháp: (<kiểu đích>)<biểu thức>. Khi đó giá trị của <biểu thức> sẽ được chuyển sang giá trị phù hợp có kiểu là <kiểu đích>. Toàn bộ biểu thức trên có giá trị là giá trị sau khi chuyển (và có kiểu là <kiểu đích>). Chẳng hạn trong khai báo với khởi tạo:

```
double f = (double)3/2;
```

thì toán tử ép kiểu bắt buộc C chuyển giá trị 3 kiểu `int` thành giá trị 3.0 kiểu `double` sau đó giá trị 2 kiểu `int` sẽ được chuyển đổi ngầm định thành 2.0 kiểu `double` để thực hiện phép chia kiểu `double`, kết quả của vế phải là 1.5. Như vậy `f` sẽ chứa giá trị 1.5. Ngược lại nếu không có phép ép kiểu (`double f = 3/2;`) thì biểu thức `3/2` sẽ có giá trị là 1 (phép chia nguyên do cả 3 lẫn 2 đều kiểu `int`) nên `f` sẽ có giá trị 1.0 (1 sẽ được chuyển ngầm định thành 1.0 kiểu `double` và bỏ vào `f`). Đó cũng là tình huống mà ta hay ép kiểu tường minh. Cũng lưu ý là toán tử ép kiểu là toán tử một ngôi và có độ ưu tiên hơn hầu hết các toán tử khác. (Ở trên thì ép kiểu trước rồi mới chia sau).

Qui tắc và cách chuyển kiểu của C là gì? Khi chuyển kiểu ngầm định thì nói chung qui tắc là chuyển lên kiểu rộng hơn gần nhất. Qui tắc này còn gọi là phép *nâng kiểu*. Theo đó với số thực thì: `float` → `double` → `long double`. Với kiểu số nguyên thì: đầu tiên các kiểu `char`, `short` đều được chuyển thành `int` sau đó nếu cần thì chuyển theo sơ đồ: `int` → `long` → `long long`[23]. Nếu có số nguyên lẫn số thực thì ưu tiên chuyển qua kiểu số thực. Với phép ép kiểu thì ta có thể yêu cầu chuyển đổi (hay trường hợp 2, 3, 4 ở trên) ngược lại chẳng hạn `double` thành `float` hay số thực sang số nguyên (nhớ là khi đó sẽ bỏ phần lẻ chứ không làm tròn). Với số nguyên thì chuyển xuống kiểu thấp hơn sẽ không được giá trị ban đầu nếu nằm ngoài phạm vi. Chẳng hạn xét các lệnh gán:

```
char c1 = 127;
char c2 = 129;
unsigned char c3 = 129;
unsigned char c4 = 257;
```

Ở lệnh gán đầu tiên, giá trị 127 kiểu `int` sẽ được chuyển thành kiểu `char` do `c1` kiểu `char`. Đây là kiểu số nguyên có dấu 1 byte với phạm vi {-128, …, 127}. Vì 127 trong phạm vi nên `c1` vẫn có giá trị là 127 (nhưng kiểu `char`). Với lệnh gán thứ 2 thì 129 nằm ngoài phạm vi nên sẽ được chuyển thành một giá trị *x* kiểu `char` nào đó (không phải là 129). *x* là giá trị nào trong các số {-128, …, 127}? Ta sẽ tìm hiểu kĩ hơn sau nhưng qui tắc bình dân là: cộng hoặc trừ một bội số của 2^n (với *n* là kích thước, theo bít, của kiểu đích) để được giá trị trong phạm vi. Ở đây vì 129 lớn hơn giá trị lớn nhất của kiểu `char` (127) nên ta trừ bớt cho 2^8 (kiểu char có kích thước là 8 bít) khi đó ta được $127 – 2^8 = 127 – 256 = -127$ thuộc phạm vi. Như vậy `c2` có giá trị là -127. Ở lệnh gán thứ 3 thì do kiểu `unsigned char` có phạm vi {0, …, 255} nên `c3` vẫn có giá trị là 129 do 129 thuộc phạm vi. Tuy nhiên ở lệnh gán thứ 4 thì giá trị `int` 257 nằm ngoài phạm vi. Vì nó lớn hơn giá trị lớn nhất 255 nên ta trừ bớt cho 2^8 được 1 trong phạm vi nên ta có `c4` chứa 1.

Thật ra còn một chỗ nữa có thể dẫn đến chuyển kiểu rất đặc biệt trong C đó là trong biểu thức điều kiện. Giả sử `i` là biến kiểu `int`, `x` là biến kiểu `double`. Bạn có biết kiểu của biểu thức điều kiện sau là gì không?

```
i > 0 ? i : x
```

Về mặt ngữ nghĩa thì nếu `i > 0` là đúng thì biểu thức trên có giá trị là `i` (kiểu `int`) còn nếu `i > 0` sai thì biểu thức trên có giá trị là `x` (kiểu `double`) như vậy tùy theo giá trị đang chứa trong `i` mà biểu thức có giá trị và có kiểu khác nhau. Như vậy giá trị cũng như kiểu của biểu thức là động, chỉ xác định lúc chạy (giá trị của `i` khi thực thi biểu thức đó) nghĩa là ta chưa thể xác định

---

[23] Thực ra cũng khá phức tạp do số nguyên có kiểu có dấu và không có dấu, ta sẽ làm kĩ hơn sau.

được giá trị và kiểu của biểu thức lúc biên dịch. Điều này đúng trong một số *ngôn ngữ kiểu động* nhưng C là *ngôn ngữ kiểu tĩnh* tức là kiểu phải được xác định lúc biên dịch. Giải pháp ở đây đó là C sẽ chọn kiểu cho biểu thức trên là `double`, lúc thực thi nếu `i > 0` sai thì giá trị là `x` (kiểu `double`) còn `i > 0` đúng thì giá trị là `i` nhưng đã được chuyển ngầm định thành kiểu `double`.

Cuối cùng, về mặt kĩ thuật, C cho phép "*định nghĩa*" kiểu "mới" bằng cách dùng `typedef` với cú pháp:

`typedef` <tên kiểu cũ> <tên kiểu mới>;

Trong đó `typedef` là từ khóa. Đây là cấu trúc khai báo một tên khác cho một kiểu đã có chứ không thực sự "tạo" ra kiểu mới. Sau cấu trúc khai báo này thì <tên kiểu mới> và <tên kiểu cũ> đều chỉ đến cùng một kiểu và có vai trò như nhau. Ta thường dùng cách này để đưa ra các tên kiểu gợi nhớ và có ý nghĩa hơn cho các dạng dữ liệu nào đó. Chẳng hạn ta có thể viết lại hàm kiểm tra tính nguyên tố (Mã 3.5.5) một cách rõ ràng, gợi nhớ hơn như sau.

Mã 3.5.7 – Hàm kiểm tra tính nguyên tố với kiểu `Bool` được `typedef`

```
1  #include <stdio.h>
2  
3  typedef int Bool;
4  
5  #define TRUE    1
6  #define FALSE   0
7  
8  Bool isprime(int n)
9  {
10     for(int i = 2; i < n; i++)
11         if(n % i == 0)
12             return FALSE;
13     return TRUE;
14 }
15 
16 int main()
17 {
18     // Nhap n >= 2
19     if(isprime(n))
20         printf("So nguyen to");
21     else
22         printf("Hop so");
23 }
```

Ta dùng `typedef` ở Dòng 3 để đưa ra một tên gợi nhớ cho kiểu luận lý là `Bool`[24] mà thật ra nó chỉ là tên khác của kiểu `int`. Qui ước đặt tên thường dùng cho các tên kiểu được `typedef` là bắt đầu bằng chữ cái hoa và sau đó

[24] Viết tắt của Boolean (nghĩa là luận lý, đặt theo tên của nhà Logic George Boole).

là chữ cái thường. Như vậy ta nên dùng tên Bool (hay Boolean) mà không nên dùng bool (hay BOLL). Thêm nữa, ở Dòng 5 và Dòng 6 ta #define hai hằng tượng trưng gợi nhớ là TRUE/FALSE cho Đúng/Sai mà thật ra chúng chính là 1/0. Cũng nhớ rằng qui ước đặt tên thường dùng cho các hằng tượng trưng được #define là toàn chữ hoa. Sau khi có các tên gợi nhớ này thì hàm isprime được viết lại dễ hiểu và rõ ràng hơn (mà thực sự đoạn mã vẫn như vậy, ta chỉ dùng các tên khác gợi nhớ hơn cho dữ liệu).

BÀI TẬP

Bt 3.5.1 Viết hàm double round(double x, int pos) để tìm giá trị làm tròn của số thực x đến chữ số ở hàng pos sau dấu chấm thập phân với qui ước hàng đơn vị có pos là 0, hàng ngay sau dấu chấm (hàng phần 10) có pos là 1, … Ví dụ: round(25.649, 0) sẽ trả về 26.0, round(25.649, 1) sẽ trả về 25.6, round(25.649, 2) sẽ trả về 25.65, … Bạn có làm được cho trường hợp pos < 0 không? Ví dụ: round(25.649, -1) sẽ trả về 30.0, round(25.649, -2) sẽ trả về 0.0, …

Bt 3.5.2 Viết hàm int digit(double x, int pos) để tìm chữ số ở hàng pos của số thực x với qui ước hàng đơn vị có pos là 0, hàng chục có pos là 1, … Ví dụ: digit(25.649, 0) sẽ trả về 5, digit(25.649, 1) sẽ trả về 2, digit(25.649, 3) sẽ trả về 0, … Bạn có làm được cho trường hợp pos < 0 không? Ví dụ: digit(25.649, -1) sẽ trả về 6, digit(25.649, -2) sẽ trả về 4, …

Bt 3.5.3 Cho hàm sau.

```
char func(char c)
{
    if(65 <= c && c <= 90)
        return c + 32;
    return c;
}
```

Hãy cho biết kết quả xuất ra của lệnh printf("%c", func('H')); và cho biết công việc của hàm func.

Bt 3.5.4 Cho đoạn mã:

```
int a = ('1' > 1) + (1.0 < 1);  //1
double b = 10.3 + '2';          //2
int c = (int)1.5 + 0.5;         //3
```

a) Giá trị của a, b, c là bao nhiêu?
b) Nêu chi tiết các bước chuyển kiểu trong các dòng mã trên (chuyển kiểu giá trị nào, từ kiểu nào sang kiểu nào, được giá trị nào).
c) Chuyển kiểu tường minh xảy ra ở những chỗ nào trong đoạn code trên? Nếu không có thì sao?

d) Theo bạn thì bước chuyển kiểu nào là đơn giản nhất và chuyển kiểu nào là phức tạp nhất trong số các bước chuyển kiểu ở trên?

Bt 3.5.5 Làm tương tự Bài tập 3.5.4 cho đoạn mã:

```
int a = 'a';                    //1
int b = (1 < a) + (a < 100);    //2
double c = 10.3 + 2;            //3
float d = (float)10/4;          //4
```

Bt 3.5.6 Cho hàm sau.

```
double fract(double x)
{
    return x - (int)x;
}
```

Hãy cho biết giá trị của v và chi tiết các bước chuyển kiểu đã xảy ra với đoạn mã `double v = fract(3/2.0);` Công việc của hàm `fract` là gì?

Bt 3.5.7 Viết hàm phù hợp cung cấp các thao tác sau trên kiểu luận lý.

a) Thao tác *XOR* 2 ngôi có bảng chân trị như sau:

Bảng 3.5.3 – Bảng ngữ nghĩa của thao tác logic *XOR*

A	*B*	*A XOR B*
Sai	Sai	Sai
Sai	Đúng	Đúng
Đúng	Sai	Đúng
Đúng	Đúng	Sai

b) Thao tác *XYZ* 3 ngôi có bảng chân trị như sau:

Bảng 3.5.4 – Bảng ngữ nghĩa của thao tác logic *XYZ*

A	*B*	*C*	*XYZ(A, B, C)*
Sai	Sai	Sai	Đúng
Sai	Sai	Đúng	Đúng
Sai	Đúng	Sai	Đúng
Sai	Đúng	Đúng	Đúng
Đúng	Sai	Sai	Sai
Đúng	Sai	Đúng	Sai
Đúng	Đúng	Sai	Đúng
Đúng	Đúng	Đúng	Đúng

Bt 3.5.8 Hãy giải thích tại sao trong đoạn mã xuất ra 128 kí tự mở rộng (có mã từ 128 đến 255) ở Bài 1.7 ta không dùng biến lặp kiểu `char` như trong đoạn mã xuất ra các kí tự ASCII in được trước đó.

Đoạn mã xuất ra các kí tự ASCII in được:

```
for(char ch = 32; ch <= 126; ch++)
    printf("%c", ch);
```

Đoạn mã xuất ra 128 kí tự ASCII mở rộng:

```
for(int ch = 128; ch <= 255; ch++)
    printf("%c", ch);
```

BÀI 3.6

Bạn đã thấy lệnh nhảy break cho phép thoát khỏi (kết thúc) một lệnh lặp (while, do-while, for) hay lệnh lựa chọn switch (Bài 2.6). Bạn cũng đã thấy lệnh nhảy return cho phép thoát khỏi (kết thúc) một hàm (Bài 3.4). Bạn sẽ thấy một lệnh nhảy "trùm" là lệnh goto, cho phép nhảy đến (chuyển đến thực thi) bất kì lệnh nào trong hàm chứa lệnh goto. Ở đây ta sẽ thấy một "dạng nhảy" khác: exit. Đây không phải là một lệnh riêng của C như các lệnh nhảy khác, exit không phải là từ khóa, nó đơn giản là một hàm trong thư viện chuẩn stdlib.h. Tuy nhiên khi hàm này được gọi thì môi trường thực thi sẽ kết thúc thực thi chương trình. Cơ chế bên dưới là do môi trường quyết định. Hàm exit có prototype là:

```
void exit(int status)
```

Hàm này nhận vào một tham số nguyên là *mã trạng thái kết thúc* trả về về cho môi trường, giá trị cụ thể với ý nghĩa tương ứng thì tùy môi trường nhưng thường thì 0 là *kết thúc thành công* (không có vấn đề gì) còn khác 0 là kết thúc không thành công (có vấn đề). Bạn cũng có thể dùng hai hằng tượng trưng được #define sẵn trong stdlib.h là EXIT_SUCCESS và EXIT_FAILURE cho mục đích tương ứng (mà thường thì chúng là 0, 1[1]). Chẳng hạn để kết thúc chương trình bình thường ta có thể gọi exit(0) hoặc exit(EXIT_SUCCESS).

Hiển nhiên khi thực thi xong hàm main thì chương trình cũng kết thúc. Hàm main thường có prototype là:

```
int main()
```

Hàm main như vậy trả về một giá trị nguyên, đó chính là giá trị trả về cho môi trường khi kết thúc chương trình. Như vậy lệnh return giá trị nguyên trong main tương đương với lời gọi exit với đối số nguyên đó. Chẳng hạn trong main khi dùng lệnh return 0; hoặc return EXIT_SUCCESS; thì chương trình kết thúc thành công. Hiển nhiên là ở các hàm khác thì return chỉ kết thúc thực thi hàm đó (callee) và chương trình tiếp tục thực thi hàm gọi (caller) nhưng khi exit được gọi (trong bất kì hàm nào) thì chương trình đều kết thúc và chuyển điều khiển về cho môi trường.

Một hàm khác (cũng trong stdlib.h) có ý nghĩa tương tự là abort với prototype là:

[1] Bạn có thể xem file stdlib.h của compiler để biết.

```
void abort()
```

Hàm này cũng kết thúc chương trình như `exit` nhưng khác biệt là: `exit` kết thúc chương trình bình thường còn `abort` *kết thúc chương trình bất thường* (một kiểu như lỗi run-time). Cơ chế cũng tùy môi trường nhưng nói chung khi chương trình kết thúc bình thường thì môi trường sẽ thực hiện các việc như giải phóng, dọn dẹp, thu hồi tài nguyên, … còn kết thúc bất thường thì có thể không.

Cho đến giờ, tất cả các biến mà bạn dùng đều được khai báo bên trong hàm (thân hàm). Chúng được gọi là các *biến cục bộ*. Tuy nhiên ta cũng có thể khai báo biến bên ngoài các hàm, và những biến này được gọi là *biến toàn cục*. Hai loại biến này khác nhau ở 3 điểm sau[2]:

- *Thời gian hoạt động*: các biến trong chương trình sẽ được *cấp phát* trên vùng nhớ mà sau đó sẽ được *giải phóng*. Khoảng thời gian giữa 2 thời điểm này được gọi là thời gian hoạt động (hay thời gian sống hay vòng đời) của biến. Ta chỉ có thể truy cập (dùng) biến trong khoảng thời gian hoạt động của nó.
    - Biến toàn cục được cấp phát ngay khi chương trình khởi động (trước cả khi hàm `main` (và do đó mọi hàm khác) thực thi) và chỉ được giải phóng khi chương trình kết thúc. Thời gian hoạt động của nó như vậy là thời gian chương trình thực thi.
    - Biến cục bộ chỉ được cấp phát khi hàm chứa nó được gọi thực thi (trước khi thực thi thân hàm) và được giải phóng ngay khi hàm kết thúc (trước khi trở về caller). Thời gian hoạt động của nó như vậy là thời gian hàm chứa nó thực thi.
- *Phạm vi truy cập* (*scope*): phần mã nguồn được phép truy cập (dùng) một biến của một khai báo được gọi là phạm vi của khai báo biến (hay biến) đó.
    - Biến toàn cục có *phạm vi file*: từ vị trí khai báo đến hết file chứa khai báo đó.
    - Biến cục bộ có *phạm vi khối*: từ vị trí khai báo đến hết khối chứa khai báo đó.
- *Khởi động giá trị*: khi các biến được khai báo thì sẽ được *khởi gán* (gán giá trị đầu tiên) theo qui định cụ thể cho từng biến.
    - Biến toàn cục được khởi động giá trị là 0.
    - Biến cục bộ không được khởi gán giá trị, như vậy ta phải đảm bảo nó được gán giá trị hợp lệ (bao gồm cả khởi tạo) trước khi dùng giá trị trong biến.

Trong C ta cũng có dạng biến đặc biệt nữa là *biến cục bộ static*. Đó là các biến cục bộ được khai báo với từ khóa `static` ở trước. Các biến này có thời

---

[2] Còn những điểm nữa mà ta sẽ tìm hiểu sau.

gian hoạt động và cách khởi động giá trị như biến toàn cục nhưng lại có phạm vi truy cập như biến cục bộ (phạm vi khối).

Cuối cùng các tham số của hàm được xem như là các biến cục bộ được khai báo ngay đầu thân hàm và được khởi động giá trị là giá trị chép từ các đối số tương ứng. Xem chương trình minh họa sau.

Mã 3.6.1 – Chương trình minh họa các loại biến

```
1  #include <stdio.h>
2  int a;
3  void func(int d)
4  {
5      int b = 5;
6      static int c = 10;
7      printf("%d %d %d %d\n", ++a, ++b, ++c, ++d);
8  }
9  int main()
10 {
11     int b = 15;
12     {
13         int d = 20;
14         func(d);
15     }
16     func(25);
17     printf("%d %d %d %d\n", ++a, ++b, ++c, ++d);
18
19     return 0;
20 }
```

Biến a khai báo ở Dòng 2 là biến toàn cục do được khai báo bên ngoài hàm. Biến `b` khai báo ở Dòng 5 là biến cục bộ của hàm `func`, biến `c` khai báo ở Dòng 6 với từ khóa `static` là biến cục bộ static của hàm `func`, tham số `d` của của `func` (Dòng 3) có thể xem như là biến cục bộ đặc biệt của `func`. Biến `b` khai báo ở Dòng 11 và biến `d` khai báo ở Dòng 13 là các biến cục bộ của hàm `main`.

Bạn không thể chạy chương trình trên vì có lỗi ngữ nghĩa ở Dòng 17. Ta đã truy cập 4 "biến" a, b, c, d:

- truy cập biến `a` là hợp lệ do biến `a` có phạm vi file nên có thể được truy cập trong mọi hàm từ sau khai báo của nó.
- truy cập biến `b` là hợp lệ do hàm `main` có một biến `b` cục bộ khai báo ở Dòng 11 có phạm vi từ chỗ khai báo (Dòng 11) đến hết khối chứa nó (thân hàm `main`), nghĩa là đến hết Dòng 20.
- truy cập biến `c` là không hợp lệ do Dòng 17 không nằm trong phạm vi của biến `c` nào cả. Lưu ý là hàm `func` có khai báo biến cục bộ `c` nhưng biến đó chỉ có phạm vi từ Dòng 6 đến hết Dòng 8.

- tương tự, truy cập biến d là không hợp lệ do Dòng 17 không nằm trong phạm vi của biến d nào cả. Lưu ý là hàm func có tham số d mà có thể xem như là biến nhưng biến đó chỉ có phạm vi từ Dòng 4 đến hết Dòng 8. Đặc biệt lưu ý là hàm main cũng có biến cục bộ d được khai báo ở Dòng 13 nhưng do có phạm vi khối nên biến đó có phạm vi từ Dòng 13 đến hết khối chứa nó, nghĩa là hết Dòng 15. Khối chứa nó bắt đầu từ dấu { ở Dòng 12 đến dấu } ở Dòng 15.

Lưu ý là một lệnh như vậy nhưng đặt ở trong thân hàm func (lệnh ở Dòng 7) là hoàn toàn hợp lệ với a là biến toàn cục và b, c, d là các dạng biến cục bộ khác nhau của hàm func. Có thể nói biến cục bộ là biến riêng của hàm hay đúng hơn là của khối. Biến toàn cục là biến chung của chương trình. Các hàm có thể dùng chung và chia sẽ biến toàn cục.

Bây giờ hãy thay lệnh ở Dòng 17 bằng lệnh truy cập biến hợp lệ sau:

```
printf("%d %d\n", ++a, ++b);
```

thì chương trình không còn lỗi. Bạn hãy chạy thử và kết quả xuất ra 3 dòng:

```
1 6 11 21
2 6 12 26
3 16
```

Hai dòng đầu do lệnh xuất ở Dòng 7 trong hàm func xuất ra còn dòng cuối do lệnh xuất mới trên trong hàm main xuất ra. Các lệnh xuất đó đều tăng giá trị (thêm 1) của các biến tương ứng rồi xuất giá trị ra. Sau đây là quá trình thực thi của chương trình:

- Trước khi hàm main thực thi thì môi trường sẽ cấp phát và khởi động giá trị 0 cho biến toàn cục a (khai báo Dòng 2) và biến cục bộ static của hàm func (khai báo ở Dòng 5).
- Khi main chuẩn bị thực thi thì môi trường sẽ cấp phát vùng nhớ cho các biến cục bộ của main là b, d mà không khởi động. Sau đó môi trường sẽ thực thi hàm main.
- Khi main thực thi thì ở Dòng 11 biến b sẽ được gán giá trị 15, sau đó main thực thi Dòng 13 thì biến d được gán giá trị 20. Sau đó hàm func được gọi với đối số có giá trị là 20 (là giá trị trong biến d của main).
- Khi func chuẩn bị thực thi thì môi trường sẽ cấp phát vùng nhớ cho các biến cục bộ của func bao gồm tham số d với giá trị khởi động là giá trị của đối số tương ứng, tức là 20, biến cục bộ b của func mà không khởi động. Lưu ý biến static c không được cấp phát mà đã được cấp phát và khởi động với giá trị 0 trước đó.
- Khi func thực thi thì ở Dòng 5 biến b được gán giá trị 5 và ở Dòng 6 biến static c được gán giá trị 10. Lưu ý là với khai báo và khởi tạo như ở Dòng 6 thì biến static sẽ được gán giá trị chỉ ở lần đầu tiên khi

hàm đó thực thi còn các lần sau đó thì lệnh gán khởi động đó sẽ không được thực thi (và như vậy biến static vẫn giữ giá trị cũ của nó). Sau đó lệnh xuất thực hiện và các biến a, b, c, d đều được tăng giá trị lên thành 1, 6, 11, 21 tương ứng và xuất ra nên ta có dòng xuất thứ nhất.

- Hàm `func` kết thúc thực thi. Môi trường giải phóng các biến cục bộ của `func` là `b`, `d` và trở về hàm `main`. Lưu ý biến `c` có phạm vi static nên không bị giải phóng (và vẫn giữ giá trị 11).
- Hàm `main` thực thi tiếp với lời gọi hàm `func` ở Dòng 16 có đối số với giá trị là 25.
- Khi `func` chuẩn bị thực thi thì môi trường sẽ cấp phát vùng nhớ cho các biến cục bộ của `func` bao gồm tham số `d` với giá trị khởi động là giá trị của đối số tương ứng, tức là 25, biến cục bộ `b` của `func` mà không khởi động.
- Khi `func` thực thi thì ở Dòng 5 biến `b` được gán giá trị 5. Phép gán khởi động ở Dòng 6 không được thực thi vì `func` đã được gọi trước đó rồi (nói nôm na dấu = khởi gán cho biến static chỉ được chạy một lần duy nhất ở lần thực thi đầu tiên của hàm chứa biến). Như vậy `c` đang giữ giá trị 11. Sau đó lệnh xuất thực hiện và các biến a, `b`, `c`, `d` đều được tăng giá trị lên thành 2, 6, 12, 26 tương ứng và xuất ra nên ta có dòng xuất thứ hai.
- Hàm `func` kết thúc thực thi. Môi trường giải phóng các biến cục bộ của `func` là `b`, `d` và trở về hàm `main`. Lưu ý biến `c` có phạm vi static nên không bị giải phóng (và vẫn giữ giá trị 12).
- Hàm `main` thực thi tiếp lệnh xuất mới đã thay vào Dòng 17 và các biến `a`, `b` (`a` là biến toàn cục đang chứa giá trị 2 và `b` là biến cục bộ của `main` (chứ không không phải của `func` là biến đã bị giải phóng rồi) đang chứa giá trị 5) đều được tăng giá trị lên thành 3, 16 tương ứng và xuất ra nên ta có dòng xuất thứ ba.
- Hàm `main` kết thúc thực thi. Môi trường giải phóng các biến cục bộ của `main` là `b`, `d`.
- Môi trường giải phóng các biến a (toàn cục) và `c` (static của `func`) và các hoạt động kết thúc bình thường khác (như đã nói ở phần `exit` trên).
- Chương trình đến đây là kết thúc rồi:)

Vì biến toàn cục là biến chung của chương trình nên nó có thể được dùng như phương tiện trao đổi thông tin/liên lạc/giao tiếp giữa các hàm bên cạnh phương tiện chính thống là tham số của hàm (và tương ứng là đối số trong lời gọi hàm). Tuy nhiên việc một hàm có dùng (đọc và/hoặc thay đổi giá trị) biến toàn cục làm cho sự thực thi hàm đó phụ thuộc vào giá trị đang có của biến và sự thực thi là có hiệu ứng lề, mà điều này thường gây khó khăn cho việc kiểm soát các hàm. Hơn nữa biết toàn cục có thể xem như là

tài nguyên chung mà việc quản lý/chia sẽ những dạng tài nguyên này luôn là vấn đề khó khăn[3]. Việc nhiều hàm cùng phụ thuộc vào biến toàn cục sẽ làm cho chúng phụ thuộc nhau mà điều này trái với nguyên tắc thiết kế là "càng ít phụ thuộc/càng nhiều độc lập càng tốt". Tuy nhiên ta sẽ không đi sâu hơn vào vấn đề này. Nói chung bạn nên *hạn chế dùng biến toàn cục trừ phi việc dùng chúng là thật sự có ý nghĩa*. Thường thì nếu một dữ liệu/thông tin mà được nhiều hàm cùng chia sẻ/xử lý thì nó có thể là biến toàn cục. Tuy nhiên không phải luôn như vậy. Cũng *tuyệt đối không dùng biến toàn cục vì lười khai báo biến cục bộ nhiều lần*. Chẳng hạn có thể bạn khai báo một biến toàn cục như `int i` để dùng nó trong các vòng lặp vì vòng lặp nào (nhất là các vòng lặp `for`) cũng thường dùng một biến như vậy. Tuyệt đối tránh nhé. Bạn sẽ thấy một trường hợp sử dụng biến toàn cục rất hay mà tôi sẽ giới thiệu sau.

Cách dùng biến cục bộ tĩnh thì khá rõ ràng. Đó là biến cục bộ (riêng) của hàm nhưng nếu ta muốn nó duy trì giá trị qua các đợt chạy sau của hàm thì đặt nó là biến tĩnh.

Một vấn đề quan trọng khi bạn có nhiều biến là vấn đề đặt tên, bạn có thể đặt tên trùng cho biến không? Chẳng hạn các biến lặp thì thường đặt là `i`, `j`, … các biến chỉ số lượng thì hay đặt là `n`, `length`, `count`, … Qui tắc về phạm vi của các biến giúp ta có thể làm được điều này. Chẳng hạn vì các biến cục bộ chỉ có phạm vi khối nên ta có thể dùng biến cục bộ trùng tên trong các hàm khác nhau. Như ở trên thì hàm `func` và hàm `main` có biến cục bộ trùng tên là biến `b`. Điều này hoàn toàn không có vấn đề gì, mỗi khai báo có phạm vi trong khối chứa nó và hai phạm vi này rời nhau. Trong `main` (chính xác hơn là trong phạm vi của khai báo ở Dòng 15) thì `b` là `b` của `main` (chính xác hơn là của khai báo ở Dòng 15), còn trong `func` thì `b` là `b` của `func`. Chúng không liên quan gì nhau, ngoài việc "tình cờ trùng tên". Tương tự, `main` và `func` còn trùng tên biến `d` (`d` là tham số của hàm `func` mà ta có thể xem như là biến cục bộ).

Tuy nhiên trường hợp trùng tên biến trong cùng một phạm vi thì sao? Sau đây là qui tắc của C để phân giải trường hợp này:

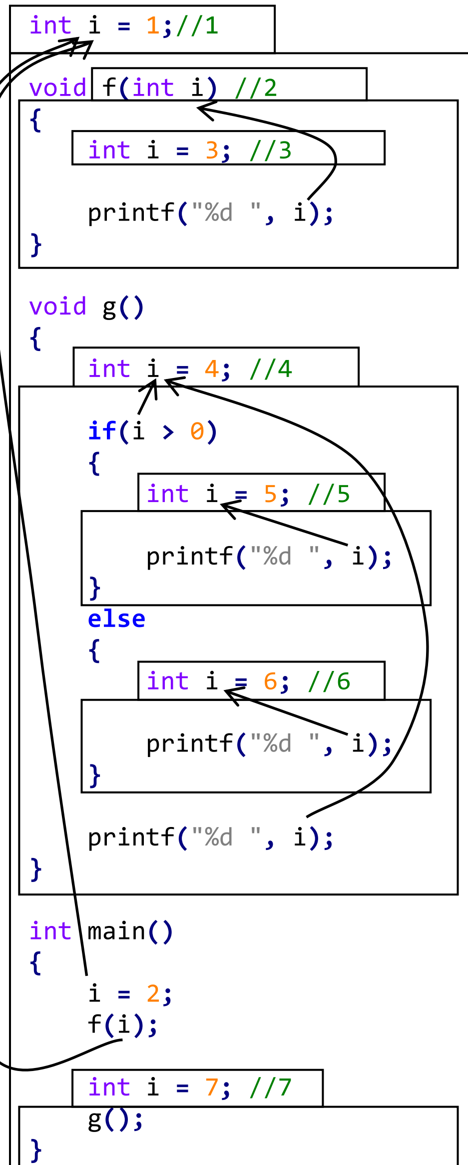
- Trong cùng một file thì không thể có hai khai báo biến toàn cục trùng tên và trong cùng một khối thì không thể có hai khai báo biến cục bộ trùng tên. Nếu xảy ra điều này thì đây là một lỗi ngữ nghĩa gọi là lỗi "định nghĩa lại".
- Khi một biến được truy xuất (dùng) thì ưu tiên cho khai báo ở các khối bên trong hơn bên ngoài: C tìm khai báo (định nghĩa) của biến ứng với một truy xuất biến bằng cách dò từ trong ra ngoài bắt đầu từ khối chứa truy xuất biến đến khối chứa khối đó rồi đến khối chứa khối chứa khối đó, … đến bên ngoài các khối (mức toàn cục trên file)

[3] Bạn đã thấy khó khăn của việc quản lý/chia sẻ tài sản công là thế nào rồi đó.

cho đến khi tìm thấy. Nếu không thì ta đã dùng một biến mà chia định nghĩa (khai báo), một lỗi ngữ nghĩa "dùng biến chưa được định nghĩa".

Như vậy C cho phép các khối lồng chứa khác nhau có thể có khai báo trùng tên biến và khai báo bên trong sẽ *che* khai báo trùng tên bên ngoài. Ví dụ sau sẽ cho thấy rõ.

Hình 3.6.1 – Ví dụ minh họa phạm vi biến với các biến trùng tên



Ta có 7 khai báo biến trùng tên là i được ghi chú từ //1 đến //7. Phạm vi của chúng được thể hiện bằng khối chữ nhật kèm theo ngay dưới. Khai báo 1 là khai báo biến toàn cục với phạm vi từ đó đến hết file (như vậy các hàm f, g, main đều nằm trong phạm vi của nó). Khai báo 2 là khai báo tham số của hàm f mà ta xem như là biến cục bộ có phạm vi toàn bộ thân hàm f. Khai báo 3 là khai báo biến cục bộ của f. Khai báo này trùng tên với khai báo 2 cùng khối là thân hàm f nên khai báo 3 không hợp lệ với lỗi ngữ nghĩa là khai báo lặp. Ta phải bỏ khai báo này đi để được chương trình hợp lệ. Truy cập biến i của lệnh xuất trong hàm f nằm trong phạm vi của cả khai báo 1 và khai báo 2 nhưng vì khai báo 2 nằm trong khối bên trong (thân hàm f) của khối chứa khai báo 1 (toàn file) nên theo qui tắc "trong nhất" thì truy cập đó tương ứng với khai báo 2, tức là lệnh xuất đó sẽ xuất ra giá trị của tham số i (mà đó chính là giá trị truyền vào của đối số tương ứng trong lời gọi hàm). Trên Hình 3.6.1 thì mũi tên từ truy xuất biến đến khai báo biến cho biết điều này. Khai báo 4 có phạm vi từ khai báo đến hết khối chứa nó là thân hàm g. Khai báo 5 có phạm vi từ khai báo đến hết khối chứa nó là thân if. Khai báo 6 có phạm vi từ khai báo đến hết khối chứa nó là thân else. Lưu ý là khối thân if và khối thân else là hai khối rời nhau và đều là khối con của (chứa trong) khối thân hàm g cho nên 3 khai báo 4, 5, 6 đều hợp lệ. Cũng theo qui tắc "trong nhất" thì các truy xuất biến trong các lệnh xuất trong thân if, else là tương ứng với khai báo 5, 6 như hình (chứ không phải khai báo 4). Do đó giá trị xuất ra tương ứng là 5, 6.

Truy cập biến i trong điều kiện của if không nằm trong thân if hay else nên nằm trong phạm vi của khai báo 4 (và 1) như vậy truy cập này tương ứng với khai báo 4. Tương tự, lệnh xuất cuối cùng của g nằm ngoài khối thân if và thân else nên nằm trong phạm vi của khai báo 4 (và 1) cho nên truy cập cũng tương ứng với khai báo 4 và giá trị xuất ra tương ứng là 4.

Trong main thì lệnh gán đầu tiên truy cập biến i trong phạm vi duy nhất của một khai báo là khai báo 1 nên truy cập đó là ứng với khai báo 1. Như vậy biến toàn cục i được đặt lại giá trị là 2. Lời gọi hàm f(i) tiếp đó thì truy cập i cũng ứng với khai báo 1 với giá trị đang chứa trong i (toàn cục - ứng với khai báo 1) là 2. Như vậy hàm f được gọi với giá trị của đối số là 2. Khai báo 7 có phạm vi từ khai báo đó đến hết khối chứa nó là thân hàm main. Lưu ý là lệnh gán i = 2; (và lời gọi hàm f(i)) đặt trước khai báo 7 nên không nằm trong phạm vi của khai báo 7. Nếu 2 lệnh này đặt sau khai báo 7 thì nó sẽ nằm trong phạm vi của khai báo 7 và do đó truy cập đến i bấy giờ sẽ là khai báo 7 chứ không phải là khai báo 1.

Bạn hãy bỏ khai báo 3 ở trên và chạy thử, kết quả xuất ra là: 2 5 4. Bạn thử bỏ các khai báo khác nhau ở trên và đổi 2 lệnh trước khai báo 7 ra sau nó, … và chạy thử để quan sát kết quả xuất ra (hoặc thông báo lỗi) để hiểu hơn nhé. Chẳng hạn nếu bỏ khai báo 2 (nghĩa là hàm f không còn tham số) và sửa lời gọi hàm trong main thành f() thì kết quả xuất ra là gì? Cũng như

vậy nhưng đem 2 lệnh trước khai báo 7 ra ngay sau nó thì kết quả xuất ra là gì? Tại sao có khác biệt đó? ...

Nhớ rằng thân hàm là một khối mà nó có thể chứa các khối con bên trong mà các khối con cũng có thể lại chứa các khối con của nó, ... Hơn nữa về mặt logic thì theo cách nào đó ta có thể coi một file như là một khối lớn mà các khối con chứa trong nó chính là các thân hàm. Tất cả các khối đều có thể chứa khai báo biến.

Một điều nữa về phạm vi biến: bạn chắc nhớ rằng biến có thể được khai báo trong phần khởi động của `for`:

```
for(int i = <A>; <B>; <C>)
        <D>
```

Lệnh trên có thể được hiểu theo nhiều cách khác nhau. Chẳng hạn đa số các compiler hiểu là:

```
{
        int i;
        for(i = <A>; <B>; <C>)
                <D>
}
```

Tuy nhiên cũng có compiler hiểu là:

```
int i;
for(i = <A>; <B>; <C>)
        <D>
```

Với cách hiểu thứ nhất thì `i` chỉ có phạm vi trong lệnh `for` thôi, còn cách hiểu thứ hai thì nó có phạm vi từ lệnh `for` trở đi đến hết khối chứa `for`. Chẳng hạn với cách hiểu thứ nhất thì đoạn mã sau sẽ xuất ra: `1 2 3 4 5 2`.

```
int i = 2;
for(int i = 1; i <= 5; i++)
    printf("%d ", i);
printf("%d", i);
```

Nhưng với cách hiểu thứ hai thì đoạn mã trên sẽ có lỗi "định nghĩa lại biến `i`" ở lệnh `for`. Với cách hiểu thứ hai thì đoạn mã sau sẽ xuất ra: `1 2 3 4 5 6`.

```
for(int i = 1; i <= 5; i++)
    printf("%d ", i);
printf("%d", i);
```

Nhưng với cách hiểu thứ nhất thì đoạn mã trên sẽ có lỗi "`i` chưa được định nghĩa" ở lệnh xuất cuối cùng.

Như vậy, cho tới giờ, cách tổ chức chương trình đơn giản trong một file như sau[4]:

Các chỉ thị `#include`
Các chỉ thị `#define`
Các định nghĩa kiểu
Các khai báo biến toàn cục
Các khai báo hàm
Định nghĩa của hàm `main`
Định nghĩa các hàm khác

Khi chương trình càng lớn/càng có nhiều hàm thì nó càng phức tạp/càng có nhiều bug. Do đó bạn cần nâng cao kĩ năng debug trong các chương trình như vậy. Phụ lục A.7 hướng dẫn cơ bản cách debug chương trình.

BÀI TẬP

Bt 3.6.1[5] Cho biết công việc của các hàm `print_one_row` và `print_all_rows`. Cho biết kết quả xuất ra khi hàm `print_all_rows` được gọi rồi phân tích, rút ra nhận xét và chỉnh sửa mã cho tốt.

```
int i;
void print_one_row()
{
    for(i = 1; i <= 10; i++)
        printf("*");
}
void print_all_rows()
{
    for(i = 1; i <= 10; i++)
    {
        print_one_row();
        printf("\n");
    }
}
```

Bt 3.6.2 Cho biết hàm `f` sau đây làm công việc gì? Giải thích.

```
void f()
{
    static int i = 1;
    printf("%d ", i++);
}
```

(*Gợi ý*: thử gọi: `f(); f(); f();` để xem kết quả kết xuất.)

---

[4] Cách tổ chức chương trình phức tạp gồm nhiều phần, viết trên nhiều file sẽ được bàn trong bài sau.
[5] Bài tập từ sách "C Programming: A Modern Approach" của K. N. King.

Bt 3.6.3 Cho biết đoạn mã sau có hợp lệ không? Nếu có hãy cho biết kết xuất và giải thích.

```
int i = 1;
int main()
{
    int j = i;
    int i = 2;
    printf("%d %d", i, j);

    return 0;
}
```

Bt 3.6.4 Cho biết đoạn mã sau có hợp lệ không? Nếu có hãy cho biết kết xuất và giải thích.

```
int main()
{
    int i = 1;
    {
        printf("%d ", i);
        int i = 2;
        {
            printf("%d ", i);
            int i = 3;
            printf("%d ", i);
        }
        printf("%d ", i);
    }
    printf("%d ", i);

    return 0;
}
```

Bt 3.6.5 Cho biết kết xuất của chương trình sau. Để ý việc dùng biến toàn cục `i` (khai báo ở Dòng 3). Nghiền ngẫm để hiểu hoạt động của chương trình qua tương tác giữa các hàm với biến toàn cục `i`. (*Gợi ý*: có "bóng dáng" của vòng lặp `for` trong chương trình.) Sửa các hàm `start`, `step`, `stop` để chương trình xuất ra các số chẵn 100, 98, ..., 2, 0.

Mã 3.6.2 – Chương trình của Bài tập 3.6.5

```
1  #include <stdio.h>
2  
3  int i;
4  
5  void start() {
6      i = 1;
7  }
8  
```

```
void step() {
    i++;
}

int stop() {
    return i > 10;
}

void act() {
    printf("%d ", i);
}

void loop() {
    if(stop())
        return;
    act(); step(); loop();
}

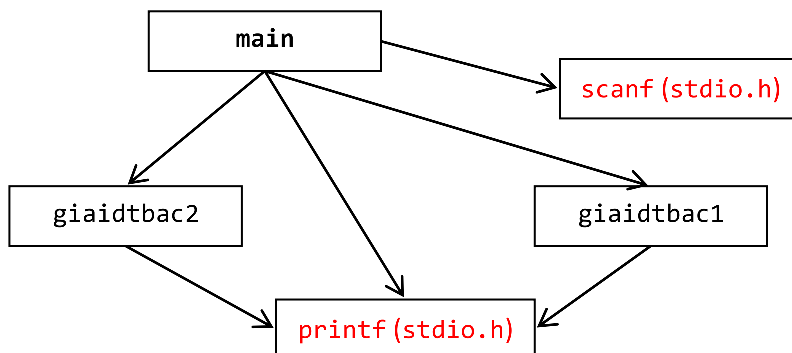
int main() {
    start();
    loop();

    return 0;
}
```

BÀI 3.7

Một chương trình C đơn giản có thể chỉ có một hàm là hàm `main`. Nhưng một chương trình phức tạp hơn có thể có nhiều hàm khác nữa. Khi đó ta cần có cách quản lý các hàm. Cơ bản nhất ta cần biết hàm nào gọi hàm nào. Khi thân hàm *A* có lời gọi đến hàm *B* ta nói *A* là *hàm gọi* (*caller*) còn *B* là *hàm được gọi* (*callee*), ta còn nói hàm *A* có dùng hàm *B*. Rõ ràng khi đó *A* sẽ phụ thuộc vào *B*, chẳng hạn nếu *B* chưa được cài đặt thì *A* sẽ không thực thi được hay *B* thực thi sai thì *A* cũng sẽ sai theo hay *A* bị lặp vô hạn thì *B* sẽ bị lặp vô hạn, … Một cách đơn giản nhất để thấy mối liên quan giữa các hàm (gọi hàm hay dùng hàm hay phụ thuộc) là dùng *đồ thị phụ thuộc hàm* hay *đồ thị gọi hàm*. Ví dụ sau đây là đồ thị gọi hàm của các hàm trong chương trình giải đa thức $ax^2 + bx + c$ (hoàn thiện Mã 3.3.1 Bài 3.3).

Hình 3.7.1 – Đồ thị gọi hàm của chương trình giải đa thức $ax^2 + bx + c$

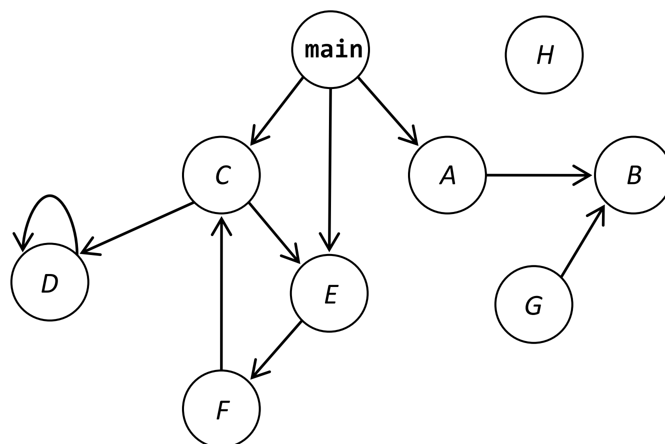


Sơ đồ trên gồm các *nút* là các hàm và các *cạnh có hướng* (mũi tên) cho biết hàm nào gọi hàm nào. Khi hàm *A* gọi hàm *B* thì ta có cạnh (mũi tên) từ nút *A* đến nút *B*. Ta còn nói đó là cạnh $A \rightarrow B$, *A* là *nút đầu* còn *B* là *nút cuối* của cạnh. Ta cũng nói đó là *cạnh ra khỏi A* và là *cạnh đi vào B*. Cạnh từ *A* đến *B* cũng cho thấy hàm *A phụ thuộc* hàm *B*. Chẳng hạn ở trên ta có 5 nút ứng với 5 hàm có xuất hiện trong chương trình. Có mũi tên từ nút `main` đến nút `giaidtbac2` cho thấy hàm `main` có gọi (và do đó phụ thuộc vào) hàm `giaidtbac2`. Tương tự các mũi tên khác cho thấy hàm `main` còn phụ thuộc vào các hàm `giaidtbac1`, `printf` và `scanf`. Hàm `giaidtbac2` và `giaidtbac1` có gọi hàm `printf`. Ở đây hai hàm `printf` và `scanf` là 2 hàm đặc biệt, chúng thuộc thư viện chuẩn của C (trong `stdio.h`), ta không biết chúng có gọi các hàm khác hay không. Thật ra ta không quan tâm vì ta

không viết các hàm này, như vậy ta không phân tích các cạnh ra khỏi nút này. Đối với các hàm trong thư viện chuẩn, đôi khi ta bỏ đi để đồ thị gọi hàm đỡ rối. Hiển nhiên nút đặc biệt nhất trong đồ thị trên là nút `main` (nút ứng với hàm `main`). Đây gọi là *nút gốc* của đồ thị vì mọi phân tích của ta đều xuất phát từ nút này. Nút này có điều đặc biệt là không có cạnh vào nó, không hàm nào được gọi hàm này[1], hàm này chỉ được môi trường gọi chạy để thực thi chương trình.

Một ví dụ đồ thị gọi hàm phức tạp hơn được cho như hình dưới.

Hình 3.7.2 – Một đồ thị gọi hàm



Ở trên ta đã vẽ nút tròn thay vì nút chữ nhật mà điều này chỉ để cho đẹp thôi. Có vài điều phân tích được từ đồ thị này:

- Hàm `main` có gọi hàm *A* và hàm *C*. Hàm *A* có gọi hàm *B*. Như vậy ta có `main` *phụ thuộc trực tiếp* vào *A*, *C*. *A* phụ thuộc trực tiếp vào *B*. Và dẫn đến `main` *phụ thuộc gián tiếp* vào *B*. Như vậy mặc dù `main` không gọi *B* (không có cạnh từ `main` đến *B*) nhưng `main` vẫn phụ thuộc *B*. Ta nhận diện điều này bằng *đường đi* (dãy cạnh liên tiếp) từ `main` đến *B* trên đồ thị. Tương tự ta có `main` phụ thuộc vào tất cả các hàm sau: *A*, *B*, *C*, *D*, *E*, *F*.
- Ta cũng thấy `main` không phụ thuộc vào hàm *G* và hàm *H* vì không có đường đi nào từ `main` đến *G* hay *H*. Điều này cũng có nghĩa là ta không cần hàm *G* và hàm *H* trong chương trình vì khi chương trình thực thi, hàm *G* và hàm *H* chắc chắn không được thực thi. Vậy nên ta có thể bỏ (xóa bỏ) hàm *G* và hàm *H* đi mà không ảnh hưởng đến chương trình.

[1] Thực ra C cho phép hàm `main` gọi hàm `main` hay các hàm khác ta viết cũng được gọi hàm `main` nhưng điều này là hoàn toàn không nên.

- Hàm *D* rất đặc biệt, nó có lời gọi đến chính nó, nó phụ thuộc chính nó. Một hàm như vậy được gọi là hàm *đệ qui* (trực tiếp). Ta nhận diện dạng đệ qui này bằng *khuyên* (cạnh từ một nút đến chính nút đó) trên đồ thị gọi hàm.
- Hàm *C*, *E*, *F* còn đặc biệt hơn: *C* gọi *E*, *E* gọi *F*, *F* gọi *C* dẫn đến *C* phụ thuộc chính nó (và *E*, *F* cũng phụ thuộc chính nó). Đây là dạng phức tạp hơn của đệ qui mà ta gọi là *đệ qui tương hỗ*. Ta nhận diện dạng đệ qui này bằng *chu trình* (đường đi bắt đầu và kết thúc tại cùng một nút) trên đồ thị gọi hàm.

Bạn có thể khó hình dung về đệ qui: hàm gọi chính nó, phụ thuộc chính nó hay được định nghĩa bằng chính nó. Hơn nữa đệ qui cũng rất dễ dẫn đến thực thi vô tận, thực thi *D* dẫn đến thực thi *D* dẫn đến thực thi *D*, … hay thực thi *E* dẫn đến thực thi *F* dẫn đến thực thi *C* dẫn đến thực thi *E*, … Tuy nhiên các hàm đệ qui lại xuất hiện rất tự nhiên và cũng thường xuyên được dùng. Sau đây là một ví dụ.

Mã 3.7.1 – Hàm đệ qui tính giai thừa

```
1 int fact(int n)
2 {
3     if(n == 0)
4         return 1;
5 
6     return fact(n - 1) * n;
7 }
```

Hàm trên giúp ta tính giai thừa của một số nguyên không âm đó (chẳng hạn gọi `fact(4)` sẽ được 24 (4! = 4×3×2×1 = 24)). Hàm này đệ qui vì nó gọi chính nó (Dòng 6). Thật đáng ngạc nhiên là nó không bị thực thi vô hạn và trả về đúng giá trị (n!). Ta sẽ tìm hiểu đệ qui sau.

Lệnh là đơn vị thực thi của C nhưng khi ta phân tích ở mức toàn bộ chương trình thì *hàm là đơn vị thực thi logic của chương trình*. Chương trình có thể xem là tập hợp các hàm với mối quan hệ “gọi hàm” giữa các hàm mà ta đã mô tả bằng đồ thị gọi hàm. Thật ra khi chương trình phức tạp hơn nữa (chẳng hạn có 50, 100, 1000 hay nhiều hàm hơn nữa với nhiều chức năng, công việc khác nhau) thì ta thường chia chương trình thành các phần nhỏ hơn. Đây chính là một dạng ứng dụng của *chiến lược chia để trị* mà ta đã biết. Mỗi phần nhỏ hơn đó được gọi là *module*[2]. Mỗi module là tập hợp các hàm (và các kiểu dữ liệu) có liên quan nhau về chức năng, công việc, nhiệm vụ. Chẳng hạn nếu ta viết chương trình tính toán như Calculator thì ta có các module như nhập/xuất quản lý việc nhập/xuất/giao diện/tương tác với người dùng, module tính toán thực hiện các thao tác tính toán, module tổ chức dữ liệu quản lý việc tổ chức dữ liệu và các thao tác cơ bản trên đó

[2] Một số ngôn ngữ gọi là namespace, package, …

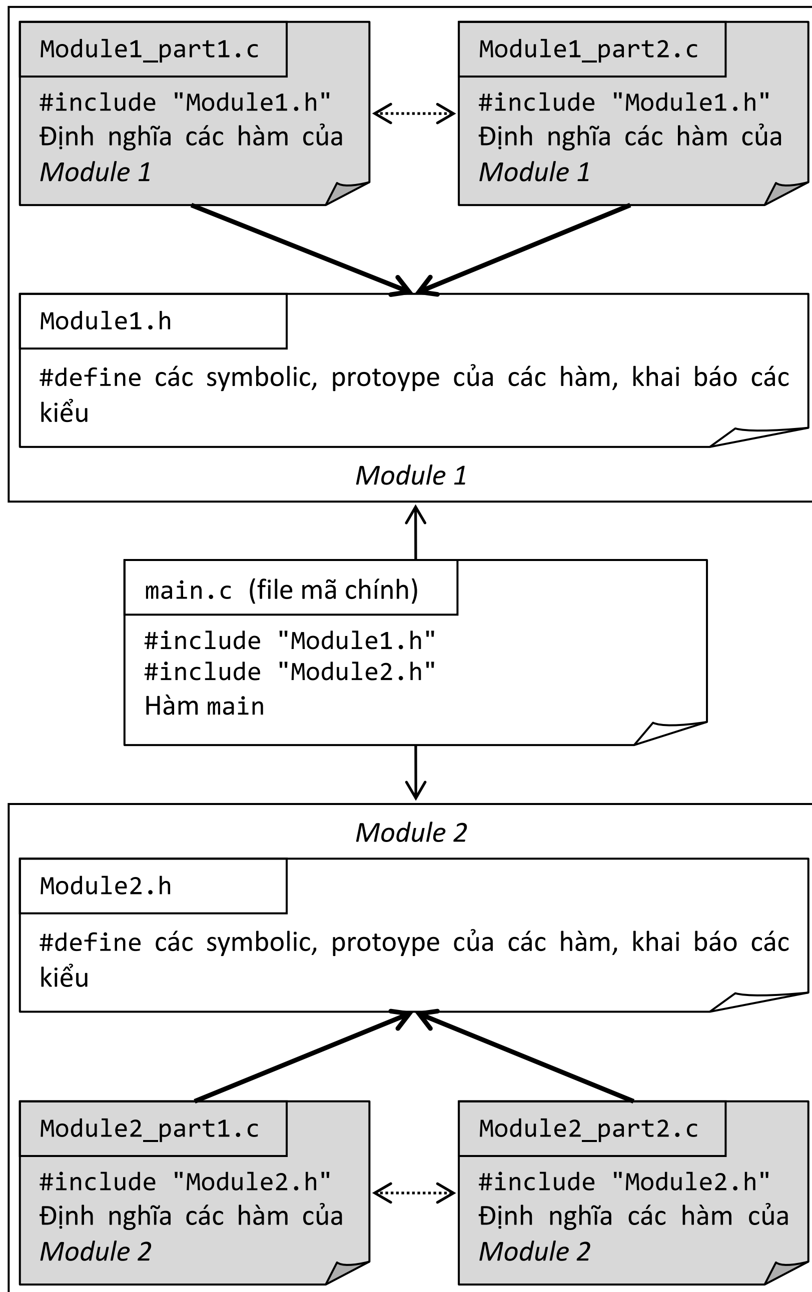
chẳng hạn khi ta muốn làm việc với số nguyên rất lớn thì rõ ràng ta không thể dùng số `int`, `long` hay thậm chí `long long` cũng không đủ, ta sẽ cần một cách tổ chức dữ liệu nào đó để làm việc với số nguyên chẳng hạn có 50, 100, … chữ số. Tất nhiên việc tổ chức cụ thể: có phân chia thành nhiều module hay không? Nếu có thì bao nhiêu? Module nào gồm có những gì? Có chia module thành các phần nhỏ hơn (module con) hay không?, … thì C không có "nhúng tay vào" (cũng như việc tổ chức hàm), những việc này do lập trình viên (thậm chí là nhóm lập trình viên khi ta viết chương trình lớn) quyết định dựa vào từng chương trình, kinh nghiệm, sở thích, …[3]. C chỉ có một hỗ trợ đó là C cho phép viết chương trình trên nhiều file khác nhau. Một file có thể chỉ có chứa một hàm nhưng cũng có thể chứa rất nhiều hàm. Khi biên dịch, C sẽ kết hợp tất cả các hàm liên quan trong các file tương ứng thành chương trình hoàn chỉnh[4]. Một cách đơn giản ta thường để các hàm trong một module trên cùng một file hoặc chia nhỏ hơn nữa: các hàm liên quan trong một phần của module sẽ được để chung trên một file. Thật ra hiếm khi chương trình thực tế lại chỉ để trên 1 file.

Cũng như khi phân tích các hàm ở mức hệ thống, ta không quan tâm đến mã (cài đặt) của hàm đó, ta chỉ quan tâm công việc hàm đó, hàm đó làm gì (what?) chứ không phải là làm như thế nào (how?), nghĩa là ta xem các hàm như là các hộp đen. Tương tự như vậy ta thường chỉ quan tâm một module gồm có những gì: tổ chức dữ liệu và tập hợp các hàm, các hàm nhận gì, trả gì và làm gì? (chứ không phải làm thế nào), tức là ta không quan tâm cách cài đặt cụ thể của module đó. C hỗ trợ ta điều này bằng cách cho phép khai báo có thể tách riêng ra khỏi cài đặt. Chẳng hạn ta có thể đặt các prototype của các hàm (khai báo của hàm) trên một file riêng còn định nghĩa hàm (cài đặt, mã của hàm) được đặt trên một file khác. Theo qui ước, các file chỉ chứa các khai báo gọi là *file tiêu đề* và có phần mở rộng là `.h` còn các file cài đặt thì gọi là *file source* (hay *file mã*) có phần mở rộng là `.c` (hay `.cpp` với C/C++). Toàn bộ chương trình dù đơn giản hay phức tạp đều có đúng một hàm `main`. File source chứa cài đặt của hàm `main` được gọi là file mã chính hay file chương trình[5] và thường thì file chương trình này rất đơn giản, nó chỉ chứa hàm `main` thôi. Hình 3.7.3 cho thấy khuôn mẫu cách thức tổ chức module/file cho chương trình.

Hình 3.7.3 – Mẫu tổ chức module/file cho chương trình

---

[3] Đây là lãnh vực của *công nghệ phần mềm*, lãnh vực liên quan đến "công nghệ lập trình" với các chương trình thực tế, lớn, phức tạp đòi hỏi nhiều người tham gia.
[4] Quá trình này sẽ được tìm hiểu kĩ hơn sau.
[5] Cách gọi dễ gây nhầm lẫn vì chương trình phải là toàn bộ.



Mỗi module gồm một file tiêu đề (thường có đuôi là `.h` và có tên là tên gợi nhớ module) và một hoặc một vài file source cài đặt (toàn bộ hoặc các phần khác nhau) của module đó. File tiêu đề cho biết những gì mà module đó có mà các module khác (gọi chung là *bên ngoài*) có thể dùng từ module

này. File này còn được gọi là *giao diện* của module vì nó cho biết thông tin về những thứ của module mà bên ngoài có thể dùng. Đó có thể là các symbolic được `#define`, prototype của các hàm hay khai báo các kiểu dữ liệu của module.

Trong khi file tiêu đề là bộ mặt hay giao diện của module thì các file cài đặt lại là chuyện nội bộ, chuyện riêng của các module. Khi bên ngoài dùng một module thì nó cần biết giao diện của module (file `.h`) nhưng không cần (và không nên/không thể) biết về nội bộ/chi tiết cài đặt của module đó. Ở hình vẽ trên, tôi đã tô đen các file source của module để nói rằng đó là hộp đen đối với bên ngoài. Mũi tên đậm trên hình từ các file cài đặt của một module đến giao diện của nó (file tiêu đề) có ý nghĩa là cài đặt. Nó được dùng để ám chỉ rằng các file source này sẽ chịu trách nhiệm *hiện thực* (thi công) những gì đã được thể hiện trong giao diện của module.

Khi một hàm trong module *A* gọi một hàm trong module *B* ta cũng nói rằng *A* có dùng *B* và do đó module *A* phụ thuộc vào module *B* và ta thể hiện bằng mũi tên như trong đồ thị gọi hàm. Ở hình trên thì các mũi tên cho thấy module chính (chứa hàm `main`) có dùng *Module 1* và *Module 2*. Các module cũng có thể phụ thuộc vào nhau. Tuy nhiên ta nên tuân theo nguyên lý thiết kế là: giữa các module càng ít phụ thuộc nhau càng tốt (nghĩa là càng độc lập nhau càng tốt).

Giữa các phần cài đặt của module cũng có thể phụ thuộc nhau nhưng đó là chuyện nội bộ của module nên được thể hiện bằng các mũi tên chấm chấm như hình trên. Như vậy phần cài đặt chi tiết của module cũng như sự phụ thuộc giữa các phần này thì bên ngoài không cần biết đến. Cũng lưu ý là trong các file source cài đặt module có thể có chứa các macro, định nghĩa kiểu và các *hàm dùng nội bộ*, những hàm này không được khai báo trong file tiêu đề nên bên ngoài không biết và không dùng được. Điều này là hợp lý vì đây là các hàm dùng nội bộ.

Một điều quan trọng về mặt kĩ thuật là khi bên ngoài dùng một module nào đó thì cần phải thêm thông tin của module đó vào file source của mình. Điều này được thực hiện bằng cách dùng chỉ thị:

`#include` <file tiêu đề của module>

để thêm thông tin của module cần dùng vào file source mà có dùng module đó. `#include` là chỉ thị tiền xử lý, công việc thực chất của nó là chèn (chép) file tiêu đề tương ứng vào file source, và như vậy ta có được thông tin của module (như khai báo kiểu hay khai báo hàm) trong file source. Ở hình trên thì file source chính (mà ta thường đặt tên là `main.c` hay `program.c` hoặc tốt hơn là một tên gợi nhớ cho chương trình) có `#include` 2 file tiêu đề của 2 module tương ứng vì nó có dùng 2 module đó. Bản thân các file source của module cũng cần phải `#include` file tiêu đề của module đó để có các thông tin public của module. Tuy nhiên các file source của module cũng có

thể khai báo riêng (private) các hằng tượng trưng được `#define`, prototype các hàm, các khai báo kiểu dùng nội bộ trong phần cài đặt đó của module đó (nghĩa là dùng riêng trong file source đó thôi).

Điều quan trọng về mặt kĩ thuật nữa là C không cần mã nguồn của tất cả các module. Mỗi module có thể được biên dịch riêng lẻ và khi biên dịch module chính (tức là chương trình) thì chỉ cần kết quả biên dịch của các module mà nó phụ thuộc chứ không cần mã nguồn của các module đó. Điều này dẫn đến khả năng viết/dùng lại/chia sẻ các module rất lớn. Chẳng hạn bạn có thể viết một module (chứ không phải chương trình) cung cấp các chức năng (các hàm) cho một lãnh vực nào đó, sau đó bạn biên dịch và bán module đó cho người khác dùng. Khi đó thứ bạn cung cấp cho người ta là file tiêu đề của module đó và kết quả biên dịch chứ không phải là file mã nguồn. Vì file mã nguồn chứa cài đặt của các hàm, … là những thứ mà bạn cần giữ bí mật riêng cho mình, vì nếu người ta biết cách cài đặt (bằng cách đọc hiểu từ file mã của bạn) thì họ sẽ không cần bạn nữa. Những thứ trong file tiêu đề chỉ là thông tin khai báo (như prototype của hàm) nên không cần bí mật. Chi tiết kĩ thuật của quá trình này sẽ được tìm hiểu sau và cũng vậy nó thuộc lãnh vực Công nghệ phần mềm.

Như vậy, cách tổ chức chương trình thành nhiều module với nhiều file khác nhau có nhiều ưu điểm:

(1) Giúp thấy rõ cấu trúc của chương trình nhờ phân chia các hàm liên quan vào một module trong một hay một vài file cài đặt.
(2) Mỗi file source có thể được biên dịch riêng lẻ, nhất là có thể được viết bởi những cá nhân hay nhóm được phân khác nhau.
(3) Các module (hay các hàm) có thể được sử dụng lại dễ dàng bởi các chương trình/người/nhóm khác.
(4) Có thể dễ dàng chỉnh sửa và phát triển chương trình do sự cô lập vào các module.

Điều nữa về mặt kĩ thuật là C cho phép các file tiêu đề `#include` các file tiêu đề khác. Chẳng hạn nếu trong một file tiêu đề ta có dùng các hằng symbolic, các macro, hay các khai báo kiểu từ các file tiêu đề khác thì ta cũng sẽ `#include` file tiêu đề đó vào file tiêu đề này. Thật ra thì về mặt vật lý C không phân biệt file `.h` hay `.c`, đó đều là file văn bản (việc phân biệt là ở mức logic do lập trình viên qui ước) và chỉ thị tiền xử lý `#include` cho phép chèn (chép) nội dung một file văn bản vào một file khác tại vị trí của chỉ thị. Như vậy bạn có thể `#include` một file `.c` vào một file `.c` khác hay mọi kiểu có thể[6].

[6] Khi điều này xảy ra thì có nguy cơ một file sẽ được `#include` nhiều lần vào file khác và khi đó có thể dẫn đến khai báo kiểu hay định nghĩa hàm lặp lại và đây là một lỗi ngữ nghĩa. Ta sẽ học cách giải quyết trường hợp này sau, khi ta học tới kiểu `struct`.

Các module C cung cấp sẵn mà mọi chương trình có thể dùng chính là thư viện chuẩn của C. Thư viện chuẩn của C như vậy gồm các module: hỗ trợ tính toán với các hàm tính toán (file tiêu đề `math.h`), hỗ trợ nhập xuất với các hàm nhập xuất (file tiêu đề `stdio.h`), hỗ trợ xử lý chuỗi với các hàm xử lý chuỗi (file tiêu đề `string.h`), … Tương tự các module mà ta cung cấp (ta viết và biên dịch) cho người khác dùng hay người khác cung cấp cho ta dùng cũng hay được gọi là *thư viện*[7].

Ta cũng đã sử dụng các module nhập/xuất hay tính toán có sẵn của C bằng cách `#include` file tiêu đề tương ứng (`stdio.h`, `math.h`, …) vào file mã. C không cần cung cấp các file cài đặt cho các hàm trong các module này, nó chỉ cần có các kết quả biên dịch sẵn đi kèm (những file này thường là `.lib` hay `.dll`) và như vậy ta đã dùng hàm `printf`, `scanf`, `sin`, `cos`, … và ta có thể biết prototype của nó bằng cách xem `stdio.h`, `math.h`, … nhưng ta không biết được người ta (người/nhóm viết thư viện chuẩn của C đi kèm với môi trường phát triển tương ứng) đã cài đặt chúng như thế nào vì không có file mã nguồn đi kèm. Bạn thử mở các file tiêu đề đó lên xem thử nhé. Bạn sẽ thấy rất rối nhưng một qui tắc nên tuân theo là ghi chú nhiều vào trong file tiêu đề. File tiêu đề như là bề mặt, hình ảnh, hợp đồng, giao kèo, … của module với bên ngoài nên có rất nhiều thứ cần mô tả như giả định khi dùng các hàm. C không có cách nào khác ngoài chuyện bạn phải khi chú nhiều (và đính kèm theo các văn bản khác như hướng dẫn sử dụng, …).

Về mặt kĩ thuật cũng lưu ý, khi bạn `#include` các file tiêu đề của thư viện chuẩn thì nên viết là:

#include <file tiêu đề của thư viện chuẩn>

còn khi bạn `#include` file tiêu đề của các module bạn tự viết thì hãy viết là:

#include "file tiêu đề của module tự viết"

Đây chẳng qua là vấn đề tìm kiếm: các file tiêu đề của thư viện chuẩn được đặt trong một thư mục đặc biệt còn file tiêu đề của các module tự viết sẽ nằm trong thư mục chung với chương trình (file mã chính). Bạn thậm chí có thể đặt đường dẫn đầy đủ của file tiêu đề trong chỉ thị `#include` khi nó nằm ở nơi nào đó khác.

Cuối cùng, về mặt kĩ thuật, C cho phép dùng từ khóa `static` trước các khai báo biến toàn cục hay khai báo hàm (và định nghĩa hàm) để nói rằng biến toàn cục (hay hàm) tương ứng là chỉ có *phạm vi trong file*, và do đó không thể truy cập (dùng) từ các file khác. Ngược lại các biến toàn cục (hay hàm) được khai báo mà không có từ khóa `static` thì có thể được truy cập từ các file khác. Rõ ràng, các thành phần (dữ liệu, hàm) được public của một module thì phải khai báo không dùng từ khóa `static` còn các thành phần riêng tư thì nên dùng từ khóa `static` khi khai báo.

---

[7] Và như vậy, thư viện của C được gọi là thư viện chuẩn.

BÀI TẬP

Bt 3.7.1 Ở phần đầu của bài học trên, ta đã thấy cách xác định sự phụ thuộc giữa các hàm qua lời gọi hàm. Cụ thể, khi thân hàm *A* có lời gọi đến hàm *B* thì hàm *A* phụ thuộc vào hàm *B*. Sự phụ thuộc này có thể nói là *tường minh*. Tuy nhiên các hàm cũng có thể phụ thuộc nhau theo cách khác, khó thấy hơn mà ta nói là phụ thuộc *không tường minh*. Một trường hợp như vậy là khi các hàm cùng truy cập/xử lý trên các biến toàn cục. Trong Bài tập 3.6.5 ta có nhiều hàm cùng truy cập/xử lý trên biến toàn cục `i`. Hàm `loop` phụ thuộc tường minh vào hàm `stop`, `act`, `step` và bản thân `loop` do lời gọi hàm trong thân của `loop`. Tuy nhiên hàm `act` cũng phụ thuộc vào hàm `start` dù không có lời gọi tường minh `start` trong thân của `act`. Lí do của sự phụ thuộc này là nếu ta gọi `start()` rồi ngay sau đó là `act()` thì kết quả xuất ra khi chạy `act` là 1 (do biến toàn cục `i` được đặt là 1 trong `start`), ngược lại nếu ta gọi `act()` mà không gọi `start()` trước đó thì kết quả xuất ra khi chạy `act` là 0 (do biến toàn cục `i` được khởi động giá trị là 0). Như vậy *hành vi* (mà ở đây là kết xuất) của `act` phụ thuộc việc `start` có được gọi trước đó hay không. Do đó `act` phụ thuộc (không tường minh) vào `start`. Tương tự, bạn hãy xác định sự phụ thuộc giữa các hàm trong Bài tập 3.6.5 và vẽ đồ thị phụ thuộc hàm cho các hàm này[8].

Bt 3.7.2 Làm lại Bài tập 3.1.4 bằng cách tổ chức module riêng làm việc với phân số.

Bt 3.7.3 Làm lại Bài tập 3.3.5 bằng cách tổ chức module riêng làm việc với thời điểm theo hệ 24 giờ.

Bt 3.7.4 Kết hợp Bài tập 3.3.6 và Bài tập 3.3.7 bằng cách tổ chức module riêng làm việc với số nguyên tố.

Bt 3.7.5 Kết hợp Mã 2.6.4 và Bài tập 2.6.7 bằng cách tổ chức module phù hợp.

Bt 3.7.6 Kết hợp Bài tập 2.7.1 và Bài tập 2.7.2, cho người dùng lựa chọn font chữ bằng cách tổ chức các module khác nhau cho mỗi font chữ.

Bt 3.7.7 (Mở rộng[9]) Trong Bài 1.7 ta đã thấy cách mã hóa cho các kí tự (và do đó văn bản) bằng các bảng mã khác nhau. Có một cách mã khác rất đơn giản nhưng hiệu quả, ra đời từ xa xưa nhưng vẫn được dùng cho đến giờ

---

[8] Bài tập này cũng cho thấy sự khó khăn khi phân tích chương trình (để hiểu hành vi của chương trình) nếu chương trình có dùng biến toàn cục. Nói chung, trừ khi có lý do chính đáng, ta không nên dùng biến toàn cục trong chương trình.

[9] Yêu cầu của các bài tập được đánh dấu "Mở rộng" là khó nhưng chúng xứng đáng để bạn bỏ công làm: bạn sẽ được một chương trình rất thú vị và công lực lập trình của bạn cũng tăng nhanh đáng kể.

trong nhiều tình huống khác nhau. Đó là *mã Morse*[10]. Bạn hãy tham khảo bài viết "Morse code" trên trang Wikipedia[11] và "mã Morse" trên trang Wikipedia Việt[12], đồng thời xem lại Bài 1.7 về chuỗi. Hãy viết chương trình (với các module phù hợp) cung cấp các chức năng sau:

(1) Dịch mã Morse ra văn bản thường.
(2) Dịch văn bản thường ra mã Morse.
(3) (Nâng cao) Điểm hay của mã Morse là ta có thể truyền và nhận tín hiệu Morse bằng các phương tiện và công cụ đơn giản nhất như âm thanh (còi thổi với độ dài cho dot, dash và các khoảng nghỉ phù hợp), ánh sáng (đèn sáng/tắt với thời gian sáng cho dot, dash và các khoảng nghỉ phù hợp), … . Hãy tìm kiếm một module (có thể đơn giản là vài hàm nào đó) cho phép chơi âm thanh (play sound) và thêm chức năng phát mã Morse bằng âm thanh.
(4) (Nâng cao) Ở chiều ngược lại, bạn có thể thêm chức năng nhận mã Morse bằng âm thanh không? (*Gợi ý*: Tìm kiếm module thu âm thanh (record sound) và xử lý âm thanh (process sound).)

Trong 2 chức năng đầu, bạn có thể dùng kí tự `'.'` cho dot, `'-'` cho dash, kí tự `' '` (space) giữa các letter và 2 space (hay 1 xuống dòng, `'\n'`) giữa các word. Tuy nhiên bạn hãy tự mình đưa ra các quyết định hợp lý. Hãy thử nghiệm và sáng tạo. Hãy dẹp "lệnh hãy" đã đeo đẳng bạn đến giờ. Đây là lúc bạn tỏa sáng:) *Hãy làm tôi và chính bạn bất ngờ*!

---

[10] Bạn có thể đã dùng mã Morse trong dịp cắm trại nào đó.
[11] https://en.wikipedia.org/wiki/Morse_code
[12] https://vi.wikipedia.org/wiki/Mã_Morse

BÀI 3.8

Làm thế nào để lập trình? Các kĩ năng nào là cần thiết để lập trình tốt? *Lập trình là viết chương trình*. Vậy làm thế nào để viết một chương trình? Các bước để viết một chương trình là gì? *Chương trình là một kịch bản thực thi* mà máy (hay hệ thống nào đó) sẽ thực thi để giải quyết/thực hiện một bài toán/công việc nào đó[1]. Công việc có thể ở nhiều mức độ: từ đơn giản như việc giải một phương trình bậc hai đến phức tạp như giải một phương trình bất kì hay thậm chí là "giải Toán". Công việc có thể rõ ràng như các bài toán Toán học hay mơ hồ, khó thấy như các công việc thực tế. Công việc cũng có thể ở các lãnh vực khác nhau. Ở đây ta sẽ gọi chung chúng là *bài toán* (hay *vấn đề* - problem). Như vậy chương trình là kịch bản giải một bài toán nào đó mà khi máy thực thi chương trình thì nó trở thành một hệ thống giải bài toán hay *hệ giải quyết vấn đề*. Do đó bước đầu tiên của việc lập trình là xác định/tìm hiểu/phân tích bài toán. Mà nói chung là xác định *đầu vào* (input) - những thứ được cho từ bài toán và *đầu ra* (output) - những thứ yêu cầu từ bài toán. Chẳng hạn bài toán "giải phương trình bậc 2" thì input là "một phương trình bậc 2" và output là "nghiệm của phương trình bậc 2 đó". Việc phân tích sẽ làm rõ hơn, chẳng hạn một phương trình bậc 2 là dạng phương trình $ax^2 + bx + c = 0$ như vậy một phương trình bậc 2 được xác định bởi 3 hệ số a, b, c và nó có tối đa 2 nghiệm. Hơn nữa ta giải phương trình phức hay thực? Nếu phức thì 3 hệ số a, b, c là 3 số phức và phương trình có đúng 2 nghiệm (có thể trùng nhau) còn thực thì 3 hệ số a, b, c là 3 số thực và có thể không có, có 1 nghiệm (kép – trùng nhau) và có 2 nghiệm thực[2].

Sau khi xác định bài toán, bước kế tiếp là tìm cách giải quyết bài toán. Cách giải quyết bài toán mà khi xác định rõ ràng sẽ là các bước thực hiện hay các công việc cần làm để tìm (tính/biến đổi, …) ra output từ input được gọi là *thuật toán*. Cũng như bài toán, thuật toán là cách gọi chung chung. Chúng cũng phong phú và đa dạng ở nhiều khía cạnh khác nhau. Chúng có thể rõ ràng như các bước giải phương trình bậc 2 mà chữ thuật toán là chính xác nhưng chúng cũng có thể mơ hồ như qui trình "giải quyết hồ sơ" hành chính mà từ thuật toán là chưa chính xác. Ta tạm gọi đó là *qui trình nghiệp vụ*. Gọi các bước/công việc để nấu một món đồ ăn như nấu cơm hay chiên trứng là thuật toán thì cũng hơi khiên cưỡng. Tuy nhiên ta sẽ gọi

---

[1] Cái máy gọi là *phần cứng* còn chương trình – kịch bản thực thi – gọi là *phần mềm*.

[2] Điều này còn được gọi là xác định phạm vi (mức độ nhỏ/lớn, cụ thể/tổng quát) của bài toán.

chung *cách giải quyết bài toán là thuật toán*. Do đó suy nghĩ/tìm/đưa ra thuật toán là bước thứ hai của việc lập trình. Bước này thường được gọi là *phân tích và thiết kế thuật toán*. Chẳng hạn với bài toán "giải phương trình bậc 2" thì ta có thuật toán để giải gồm các bước như trong Bài 1.5.

Khi đã xác định rõ ràng thuật toán thì bước tiếp theo là *hiện thực* hay *cài đặt* thuật toán bằng một ngôn ngữ lập trình nào đó mà ở đây là ngôn ngữ C. Công việc này sẽ đơn giản nếu bước phân tích và thiết kế thuật toán trước đó được làm tốt. Chẳng hạn nếu ta đã thiết kế các module với các hàm cụ thể, công việc của từng hàm với các bước cụ thể thì việc cài đặt chỉ là mô tả hay ghi xuống bằng ngôn ngữ lập trình mà thôi. Hơn nữa nếu đã có lưu đồ hay mã giả của thuật toán (hay thậm chí là của hàm) thì việc cài đặt là cực kì đơn giản. Trong bước cài đặt ta cũng thường dùng những thứ có sẵn của ngôn ngữ, của môi trường thực thi, của các đối tác, của cộng động, … Chẳng hạn các hàm đã có sẵn như các hàm trong thư viện chuẩn của C hay các hàm của môi trường thực thi (như ở phần Mở rộng 1.7, ta đã dùng hàm có sẵn của Windows). Ta cần kĩ năng tra cứu và tái sử dụng hợp lý những thứ hỗ trợ sẵn này. Ở bước cài đặt này ta cũng thường kiểm tra để đảm bảo cài đặt của ta không có lỗi và thỏa yêu cầu đặt ra. Chẳng hạn các hàm phải không có lỗi biên dịch, lỗi runtime hay lỗi logic. Do đó ta cần kĩ năng tìm/phát hiện và sửa lỗi mà quan trọng nhất là kĩ năng *debug*.

Với những chương trình thương mại (tức là chương trình cho các *bài toán thương mại*) mà ta hay gọi là phần mềm thì có thêm những bước sau đó nữa như *kiểm thử phần mềm*[3] (để xem chúng hoạt động ra sao, có đáp ứng được yêu cầu hay có hạn chế/lỗi gì khi hoạt động thực tế hay không), *bảo trì phần mềm*[4] và phát triển/tiến hóa phần mềm. Hơn nữa khi đó bài toán thường rất lớn nên cần phải làm nhóm và tiêu tốn nhiều thời gian. Khi đó tất cả các bước trên đều trở nên phức tạp với nhiều người tham gia và tạo nên cái gọi là *qui trình phát triển phần mềm*[5] và là đối tượng nghiên cứu của *công nghệ phần mềm*[6].

Ở đây ta tập trung hơn vào bài toán mang tính Toán hay khoa học[7], đó là những bài toán rõ ràng với các thuật toán cũng rõ ràng mà ta có thể gọi là các *bài toán học thuật* (hay hàn lâm) và là đối tượng nghiên cứu của *khoa học máy tính*[8]. Trong phạm vi của bí kíp này, ta lại chỉ xét những bài toán đơn giản mà một người có thể làm trong vài tuần (như một đồ án) hay vài ngày, vài giờ hay vài phút (như các bài tập trong sách này). Hơn nữa đây là

---

[3] Softwarte testing/verification.
[4] Software maintenance.
[5] Software development process.
[6] Software engineering.
[7] Gọi chung là bài toán học thuật.
[8] Computer science.

bí kíp lập trình C nên ta sẽ chỉ làm bước mô tả hay cài đặt thuật toán bằng ngôn ngữ lập trình cụ thể là C/C++. Các bước khác bạn sẽ học ở tài liệu khác (mà có thể lại gặp tôi ở bí kíp nào đó khác) và học qua thực tế khi các bạn làm việc/kiếm tiền thiệt.

Thật ra khoa học máy tính đặt nặng vấn đề lý thuyết hơn, chẳng hạn phân tích và thiết kế thuật toán. Ở đây ta tập trung nhiều vào bước: cài đặt/hiện thực thuật toán[9]. Bước này được gọi một cách bình dân là *coding* (*viết mã*) trong khi toàn bộ các bước được gọi là viết chương trình hay *lập trình* (*programming*) trong khoa học máy tính hay phát triển phần mềm trong công nghệ phần mềm. Khác biệt giữa viết mã và lập trình cũng như khác biệt giữa chép tranh và vẽ tranh. Khác biệt giữa người viết mã (codder) và người lập trình (programmer) cũng như khác biệt giữa thợ chép tranh và họa sĩ. Người họa sĩ thì đặt nặng bước một và hai (sáng tác) hơn còn thợ chép tranh thì là bước ba.

Viết mã là việc bạn mô tả thuật toán (đã xác định bằng ngôn ngữ tự nhiên, mã giả hay flowchart) bằng ngôn ngữ lập trình. Kĩ năng này bao gồm các kĩ năng nhỏ hơn:

1) Viết lại mã giả, lưu đồ bằng mã C.
2) Viết mã C cho công việc với thuật toán đơn giản hay không có thuật toán (thường gọi là cài đặt ngây thơ - naïve) + phân tích sơ lược để thấy độ phức tạp/hiệu quả/ cải tiến đơn giản.
3) Đọc hiểu mã C (để viết lại, phân tích phát hiện lỗi hay cải tiến, …), chẳng hạn mô tả mã giả hay lưu đồ cho mã C.
4) Phân tích/cải tiến cài đặt.

Để minh họa, ta xét bài toán kiểm tra số nguyên tố: “cho số nguyên $n > 1$, hỏi n có là số nguyên tố hay không?”

Đầu tiên, xác định bài toán:

- Input: số nguyên $n > 1$, nghĩa là $n \in \{2, 3, 4, 5, …\}$.
- Output:
  - Yes (True/Có/Đúng): nếu n là số nguyên tố.
  - No (False/Không/Sai): nếu n không là số nguyên tố (n là hợp số).

Phân tích (từ định nghĩa của số nguyên tố): một số nguyên $n > 1$ được gọi là số nguyên tố khi nó chỉ chia hết cho 2 số nguyên dương là 1 và chính nó. Nói cách khác nó có đúng 2 ước số nguyên dương. Ngược lại (có nhiều hơn 2 ước số nguyên dương) thì nó là hợp số. Một số nguyên dương n được gọi là chia hết cho số nguyên dương b khi n chia b không dư, tức số dư khi chia n cho b là 0. Chẳng hạn 11 là số nguyên tố vì nó chỉ có 2 ước số nguyên dương là 1 và 11 còn 9 là hợp số vì nó có 3 ước số nguyên dương là 1, 3 và 9.

---

[9] Cũng không dễ tách rời 3 bước: xác định bài toàn, phân tích thiết kế thuật toán và hiện thực. Tuy nhiên 2 bước đầu ta sẽ tìm hiểu ở tài liệu khác.

Hiển nhiên là các ước nguyên dương có thể có của một số nguyên dương *n* là từ 1 đến *n*. Từ đó ta có thuật toán kiểm tra số nguyên tố bằng cách đếm ước số nguyên dương rất tự nhiên (và ngây thơ – naïve) và có thể được cài đặt đơn giản như sau.

Mã 3.8.1 – Hàm kiểm tra tính nguyên tố (phiên bản 1 - ngây thơ)

```
int isprime(int n)
{
    int ndivs = 0;
    for(int i = 1; i <= n; i++)
        if(n % i == 0)
            ndivs++;

    return (ndivs == 2);
}
```

Hàm `isprime` kiểm tra số nguyên `n` có là số nguyên tố hay không với giả định `n` không quá lớn (trong phạm vi của kiểu `int`) và `n` > 1. Hàm này trả về 1 (Đúng) khi `n` là số nguyên tố và 0 (Sai) khi `n` là hợp số. Hàm trên cài đặt đúng như thuật toán (định nghĩa) đã nói: biến `ndivs`[10] chứa số lượng ước nguyên dương của `n`. Vòng lặp `for` cho phép duyệt qua tất cả các số `i` từ 1 đến `n` mà nếu số dư khi chia `n` cho `i` là 0 (nghĩa là `n` chia hết cho `i`) thì ta tăng biến đếm `ndivs` thêm 1 đơn vị. Như vậy bằng cách khởi động `ndivs` là 0 trước vòng lặp thì sau vòng lặp ta có `ndivs` chứa số lượng ước số nguyên dương của `n`. Sau đó nếu `ndivs` là 2 thì `n` là số nguyên tố còn không thì `n` không là số nguyên tố (hợp số).

Sau khi kiểm tra qua cài đặt trên bằng cách chạy thử vài giá trị như `isprime(2)`, `isprime(3)`, `isprime(4)`, … hay với `n` = 11, `n` = 9 ta thấy cài đặt trên là tốt. Dĩ nhiên ta không thể kiểm hết tất cả các trường hợp của `n` nhưng nên kiểm qua các trường hợp đơn giản, nhất là những trường hợp đặc biệt (như `n` = 2). Lưu ý rằng không được gọi `isprime(1)`, `isprime(0)` hay gọi `isprime` với đối số là số âm. Trường hợp như vậy, hàm `isprime` có quyền hành xử tùy ý vì input không đúng giả định. Những việc kiểm tra như vậy do bên ngoài phụ trách (chứ không phải hàm `isprime`). Chẳng hạn trong hàm `main`, sau khi nhập giá trị *n* từ người dùng thì ta kiểm tra các trường hợp của *n*, chỉ khi *n* > 1 mới gọi hàm `isprime`.

Thực sự thì hàm trên cài đặt hoàn toàn đúng và làm được công việc như yêu cầu. Tuy nhiên ta chưa nên bằng lòng mà dừng lại. Việc phân tích thêm một chút có thể giúp cải thiện cài đặt (ở đây là rút ngắn thời gian thực thi). Rõ ràng thời gian thực thi phụ thuộc vào số lần thực hiện vòng lặp cho nên ta tìm cách giảm số lần lặp. Ta thấy rằng hiển nhiên là số nguyên dương *n*

[10] Viết tắt của number of divisors, nghĩa là số lượng ước số.

luôn chia hết cho 1 (*n* chia 1 được *n* dư 0) và chính nó (*n* chia *n* được 1 dư 0) nên ta "đếm sẵn" 2 ước này như sau.

Mã 3.8.2 – Hàm kiểm tra tính nguyên tố (phiên bản 2 - vẫn ngây thơ)

```
int isprime(int n)
{
    int ndivs = 2;
    for(int i = 2; i < n; i++)
        if(n % i == 0)
            ndivs++;

    return (ndivs == 2);
}
```

Thực sự thì vòng lặp đã ít đi 2 lần thực thi, tuy nhiên nói rằng đã tăng tốc độ thực thi thì rất khiên cưỡng, khi n kha khá thì lặp n – 2 lần cũng không nhanh hơn lặp n lần là mấy. Tuy nhiên ý tưởng này giúp ta nhận ra rằng n luôn có ít nhất 2 ước số và như vậy chỉ cần có thêm một ước nữa trong khoảng từ 3 đến n – 1 thì số ước số sẽ lớn hơn 2 và như vậy không phải là số nguyên tố. Do đó trong vòng lặp trên chỉ cần gặp thêm một ước của n thì ta đã kết luận ngay n không là số nguyên tố mà không cần kiểm tra thêm nữa. Từ đó ta có cài đặt cải tiến sau.

Mã 3.8.3 – Hàm kiểm tra tính nguyên tố (phiên bản 3 - cải tiến)

```
int isprime(int n)
{
    for(int i = 2; i < n; i++)
        if(n % i == 0)
            return 0;

    return 1;
}
```

Cải tiến này thực sự là cách mạng:) Chẳng hạn với số n = 100000000 (1 trăm triệu) thì phiên bản 1 phải lặp, kiểm tra 1 trăm triệu lần (hay ít hơn 2 lần với phiên bản 2) nhưng ở cài đặt trên thì ngay lần lặp đầu tiên với `i` = 2 thì hàm đã kết thúc với kết quả là không nguyên tố (n chia hết cho 2)[11]. Trong trường hợp n là số nguyên tố thì thời gian thực thi phiên bản 3 và phiên bản 2 là như nhau. Chẳng hạn chạy thử với n = 100000007 (một trăm triệu lẻ 7).

Các cải tiến ở trên thật sự là cải tiến cài đặt, tức là vận dụng cài đặt khác nhau nhưng thuật toán là không đổi. Ta đã không dùng thêm kiến thức Toán nào mà sự phân tích hướng về mã. Những cải tiến như vậy bạn phải làm được trong bí kíp này vì đó là những cải tiến đơn giản. Nói tóm lại bạn nên

[11] Bạn chạy thử phiên bản 3 với phiên bản 2 và bấm đồng hồ đo nhé!:)

có cài đặt gọn gàng, hoàn chỉnh cho một thuật toán đã có. Những cải tiến sâu hơn phải là cải tiến bản thân thuật toán. Điều này đòi hỏi những phân tích, thiết kế, sáng tạo, … nằm ngoài phạm vi của bí kíp này hay vượt quá kĩ năng viết mã. Chẳng hạn bạn có biết rằng nếu *n* là hợp số thì nó phải chia hết cho số nguyên dương *a* với $1 < a \le \sqrt{n}$. Điều này thật ra không quá khó thấy với Toán: nếu *n* chia hết cho số nguyên dương *a* thì có số nguyên dương *b* để *n* = *ab* khi đó *a*, *b* không thể cùng > $\sqrt{n}$ được vì như vậy *ab* sẽ lớn hơn *n* nên một trong *a*, *b* phải ≤ $\sqrt{n}$. Điều này cho thấy trong phiên bản 3, ta không cần phải duyệt đến *n* – 1 mà chỉ cần duyệt đến $\sqrt{n}$. Như vậy ta lại có cải tiến nữa giúp tăng đáng kể tốc độ thực thi như sau.

Mã 3.8.4 – Hàm kiểm tra tính nguyên tố (phiên bản 4 - cải tiến thêm)

```
int isprime(int n)
{
    for(int i = 2; i <= sqrt(n); i++)
        if(n % i == 0)
            return 0;

    return 1;
}
```

Cải tiến này giúp giảm thời gian kiểm tra khi n là số nguyên tố chẳng hạn với n = 100000007 thì với phiên bản 3 ta phải duyệt 100000005 lần còn với cải tiến mới ta chỉ cần duyệt $\sqrt{100000007}$ = 10000 lần. Nếu bạn không làm được điều này thì không đáng trách vì đây đòi hỏi phân tích Toán (mà cụ thể là cái nhận xét – định lý trên). Tuy nhiên nếu bạn thấy được cải tiến này (vận dụng được định lý trên) mà cài đặt như Mã 3.8.4 thì thật đáng trách! Thực sự, việc tìm ra chặn trên ($\sqrt{n}$) là tốt nhưng cài như Mã 3.8.4 là hết sức bậy:

- Điều kiện `i <= sqrt(n)` của `for` sẽ được kiểm tra mỗi lần lặp. Như vậy mỗi lần lặp ta lại gọi hàm `sqrt` để tính căn n mà ta đã biết rằng việc tính căn là rất tốn thời gian. Hiển nhiên là ta chỉ cần tính căn một lần (trước vòng lặp `for`) rồi dùng lại giá trị này.
- Điều nữa mà chỉ những người cài đặt (như tôi/bạn) mới thấy đó là vấn đề số thực trên máy có thể làm cho cài đặt trên sai. Ta đã biết rằng làm việc với số thực luôn có thể có sai số (mà việc tính căn cũng chỉ là tìm giá trị gần đúng với sai số đủ nhỏ mà bạn đã biết). Điều gì xảy ra nếu n = 49 mà `sqrt(49)` lại trả về giá trị gần đúng là 6.999999 (sai số 10^{-6}). Do việc cắt cụt khi chuyển từ số thực sang số nguyên mà ta sẽ kiểm tra với `i` từ 2 đến 6. Và như vậy cài đặt trên sẽ báo rằng 49 là số nguyên tố mà thực ra nó là hợp số với 7 là một ước số mà ta đã không kiểm tra tới.

Lưu ý đến hai điều phân tích trên, ta có cài đặt không bậy bạ như sau.

Mã 3.8.5 – Hàm kiểm tra tính nguyên tố (phiên bản 5 – không bậy bạ?)

```
1 int isprime(int n)
2 {
3     int sqrtn = sqrt(n) + 1;
4     for(int i = 2; i <= sqrtn; i++)
5         if(n % i == 0)
6             return 0;
7 
8     return 1;
9 }
```

Ví dụ này cũng cho thấy không phải cứ giỏi Toán là xong. *Có một thuật toán hay cũng cần một cài đặt tốt*. Hơn nữa bạn phải chi li từng tí (phẩm chất mà những người cài đặt phải có nhưng những người làm lý thuyết, làm Toán (ở đây là phân tích/thiết kế thuật toán) thường thiếu). Cài đặt trên vẫn còn vấn đề! Đố bạn là gì, sửa lại làm sao? Gợi ý: chạy thử `isprime(2)` sẽ biết. Lưu ý, hàm `isprime` có giả định là n > 1. Đó cũng là lý do mà bạn phải kiểm tra chương trình sau khi viết. Thực sự thì ta chỉ có thể tin tưởng một chương trình (hay một hàm, một đoạn mã, một lệnh) khi ta đã chạy thử nó.

Rõ ràng, giữa thực tế cài đặt và lý thuyết thuật toán có quan hệ mật thiết nhưng chúng cũng có những đặc trưng riêng, đòi hỏi những kĩ năng riêng[12]. Tuy nhiên, với vai trò của người cài đặt, bạn cũng nên biết và có những phân tích về Toán. Thật vậy, tôi đã thấy những người cài đặt "tày máy" sửa vòng lặp `for` ở Dòng 4. Trong cài đặt trên ta đã cho `i` chạy từ 2 đến $\lfloor\sqrt{n}\rfloor + 1$[13]. Vậy nếu ta cho `i` chạy ngược từ $\lfloor\sqrt{n}\rfloor + 1$ về 2 như mã sau thì sao?

Mã 3.8.6 – Hàm kiểm tra tính nguyên tố (phiên bản 6)

```
int isprime(int n)
{
    int sqrtn = sqrt(n) + 1;
    for(int i = sqrtn; i > 1; i--)
        if(n % i == 0)
            return 0;

    return 1;
}
```

Không có vấn đề gì với tính đúng đắn cả. Tuy nhiên thời gian thực thi thì khác. Trong trường hợp n nguyên tố thì thời gian thực thi của hai cài đặt (phiên bản 6 và phiên bản 5) là như nhau. Một vài trường hợp thì việc kiểm ngược từ $\sqrt{n}$ sẽ nhanh hơn, chẳng hạn với n = 49 thì ta dừng ngay với `i` = 7,

[12] Nghe như trích dẫn từ một giáo trình triết nào đó về mối quan hệ biện chứng... :)

[13] Kí hiệu $\lfloor x \rfloor$ chỉ số nguyên sau khi cắt bỏ phần lẻ của số thực (chính là thao tác chuyển số thực thành số nguyên của C).

còn chạy xuôi từ 2 thì ta tốn thời gian chạy tới 7. Tuy nhiên trong đa số trường hợp thì việc kiểm xuôi sẽ nhanh hơn, chẳng hạn với n = 50 thì ta dừng ngay với n = 2 còn chạy ngược thì phải chạy từ 8, 7, 6, đến 5. Vậy cái nào tốt hơn? Lần nữa kiến thức Toán mà ở đây là *Lý thuyết xác suất* sẽ giúp ta trả lời. Với *n* là một số nguyên dương ngẫu nhiên, xác suất *n* chia hết cho 2 là bao nhiêu? 1/2 vì có 2 trường hợp đồng khả năng là chia dư 0 (chia hết) và chia dư 1. Xác suất *n* chia hết cho 3 là 1/3 vì có 3 trường hợp đồng khả năng là chia dư 0 (chia hết), chia dư 1 và chia dư 2. Vậy khi *i* càng lớn thì khả năng (xác suất) *n* chia hết cho *i* sẽ càng nhỏ. Vậy cài đặt trước (kiểm tra với *i* nhỏ trước) sẽ cho khả năng dừng sớm hơn cài đặt sau (kiểm tra với *i* lớn trước). Điều đó có nghĩa là cài đặt đầu sẽ nhanh hơn cài đặt sau trong nhiều trường hợp của *n*. Do đó cài đặt đầu thì tốt hơn. Câu hỏi tốt hơn mức nào (cụ thể bao nhiêu phần trăm trường hợp tốt hơn, …) thì vượt quá phạm vi của tài liệu này.

Ta đã có một cài đặt tốt cho đến giờ. Nhưng giả sử bạn muốn kiểm tra một con số rất lớn chẳng hạn như số 1,111,111,111,111,111,111 có là số nguyên tố không thì cài đặt trên không dùng được. Hiển nhiên là con số lớn như vậy thì vượt quá phạm vi của kiểu `int` (2^{32}) nhưng nếu như ta có kiểu "số nguyên lớn" (kiểu `long long` (2^{64}) hay thậm chí kiểu số nguyên nào đó lớn hơn nữa) thì cài đặt trên vẫn không dùng được do thời gian chạy quá lâu. Tại sao? Không phải do cài đặt mà do thuật toán cũng không dùng được. Ta cần phải thay đổi từ bản chất (thay đổi thuật toán hay thậm chí thay đổi phương pháp/chiến lược/hướng tiếp cận). Kiểm tra tính nguyên tố là bài toán rất quan trọng trong mã hóa mà là nền tảng của bảo mật thông tin ngày nay nên hiển nhiên là thuật toán như vậy rất cần thiết. Tuy nhiên đó là chuyện của "người khác", ta chủ yếu là cài đặt thôi mà. Và tôi cũng không ép người quá đáng nữa:)

Để minh họa khác biệt Toán với Tin, thực tế cài đặt với lý thuyết thuật toán ta xét bài toán dân gian: "Vừa gà vừa chó / Bó lại cho tròn / Ba mươi sáu con / Một trăm chân chẵn / Hỏi mấy gà, mấy chó?". Cách làm của Toán mà ai cũng biết: Gọi *x* là số gà, *y* là số chó (*x*, *y* là số nguyên dương) ta cần giải hệ: $x + y = 36$, $2x + 4y = 100$. Bằng các phương pháp giải hệ đơn giản như thế, cộng, … ta sẽ giải ra *x*, *y*. Chẳng hạn từ phương trình 1 ta có $y = 36 - x$, thế vào phương trình 2 ta có $2x + 4(36 - x) = 100$ nên $x = 22$, $y = 14$. Đáp số: 22 gà và 14 chó[14].

Tuy nhiên nếu bạn không rành Toán hay rành giải hệ thì sao? Sau đây là cách làm bằng Tin.

---

[14] Thậm chí, cách giải hay hơn của Triết là: giả sử các con gà đều biến thành chó, khi đó 36 con có 36×4 = 144 chân, như vậy dư ra 144 – 100 = 44 chân, mà mỗi con gà khi biến thành chó sẽ có thêm 2 chân, nên đã có 44/2 = 22 gà và 36 – 22 = 14 chó. Tuyệt!

Mã 3.8.7 – Hàm giải bài toán dân gian "gà chó"

```
void gacho()
{
    for(int x = 1; x < 36; x++)
    for(int y = 1; y < 36; y++)
        if((x + y == 36)
                && (2*x + 4*y == 100))
            printf("So ga: %d, so cho: %d.\n", x, y);
}
```

Vì *x*, *y* nguyên dương và *x* + *y* = 36 nên *x*, *y* chỉ có thể nhận giá trị là {1, …, 35}. Do đó ta thử tất cả các trường hợp của *x*, *y* và kiểm tra điều kiện (tổng số con và tổng số chân). Cách làm này được gọi "*vét cạn*" mà đôi khi được gọi là cách làm ngây thơ (naïve) hay cách làm trâu bò[15]. Nói là trâu bò vì ta đã tận dụng sức mạnh và sự bền bỉ cũng như tốc độ tính toán của máy, nói là ngây thơ vì ta chẳng cần dùng tí não nào:) Để tìm, vậy thì ta cứ tìm thôi. Ta thử và kiểm tra tất cả các trường hợp nên phương pháp này còn được gọi là phương pháp thử sai. Vì ta làm đại nên phương pháp này không có vẻ gì là thuật toán. Điều quan trọng là trong nhiều tình huống ta không có cách làm như kiểu thuật toán (ta không nghĩ ra được hệ hay không giải được hệ), khi đó, nó có thể lại là cách làm duy nhất. Bạn phải cài đặt được những cách làm ngây thơ (không đòi hỏi thuật toán tinh vi) như thế này. Điều này có nghĩa là bạn phải vận dụng tốt các cấu trúc của ngôn ngữ C mà ở trên là vận dụng vòng lặp lồng (2 vòng `for` hay nhiều vòng `for` lồng như trên là khuôn mẫu thường gặp).

Câu hỏi: ta có thể cải tiến để thời gian thực thi nhanh hơn không? Câu trả lời: không cần. Ta có 35×35 = 1225 lần lặp kiểm tra ở trên, mà con số này thì quá nhỏ với sức tính toán của máy. Chính xác thì nó tìm ra lời giải theo cách cài trên (gọi hàm `gacho`) trong chưa đầy nháy mắt. Vậy thì ta phải suy nghĩ để cải tiến làm gì cho mệt? Rửa được rồi!:)

Một kĩ năng quan trọng khác là *tìm kiếm và tra cứu tài liệu*. Tra cứu để hiểu rõ bài toán, các khái niệm, định nghĩa (bài toán Toán) hay các qui tắc nghiệp vụ, các vấn đề đặc thù theo lãnh vực của bài toán. Tra cứu để tìm thuật toán (có thể là các mô tả không hình thức bằng lời, hình ảnh, ý tưởng hoặc hình thức hơn như mã giả, lưu đồ, …) hay ý tưởng để hình thành thuật toán. Tra cứu để thấy cách cài đặt phổ biến (nghĩa là cách các chuyên gia, người kinh nghiệm, hay người khác, … cài đặt). Tra cứu các hàm, module, thư viện, … có sẵn như thư viện chuẩn của C, thư viện của môi trường thực thi như Windows hay thư viện của người khác, … Điều này giúp ta *tái sử dụng mã* tốt và giúp tiết kiệm thời gian cũng như công sức khi cài đặt chương trình.

[15] Mà trong bài toán này thì gọi là cách làm "gà chó" cũng được:)

Sau khi cài đặt xong (và trong quá trình cài đặt) ta cần đảm bảo mã viết ra là tốt. Thế nào là mã tốt? Trước hết nó phải đúng và sau đó là các tiêu chí khác như hiệu quả, đơn giản, đẹp, … Thế nào là mã đúng? Là mã không có lỗi. Ta đã biết rằng hầu như mọi chương trình đều có lỗi nào đó nhất là những chương trình phức tạp. Tuy nhiên ta cần phải giảm thiểu tối đa lỗi, đặc biệt là những lỗi rõ ràng (xem lại Bài 1.6). Các lỗi biên dịch thì dễ dàng bị loại bỏ nhưng các bug thì khó hơn. Xem Phụ lục A.7 để rèn thêm kĩ năng debug.

Tóm lại để trở thành một lập trình viên giỏi thì trước hết ta phải rèn luyện các kĩ năng viết mã, tra cứu, tái sử dụng mã và debug. Sau đó ta cũng cần rèn luyện thêm kĩ năng phân tích và thiết kế thuật toán. Kĩ năng phân tích, vận dụng Toán tối thiểu; kĩ năng sưu liệu mà ở mức đơn giản là comment, log, ...; kĩ năng giao lưu, học hỏi kinh nghiệm viết mã, lập trình trên các forum, trang web, … hay tham gia vào các nhóm, cộng đồng viết mã, … cũng quan trọng. Kĩ năng đọc hiểu tiếng Anh sẽ giúp tra cứu tốt và cùng với kĩ năng làm việc nhóm sẽ chuẩn bị cho bạn khả năng tham gia vào các nhóm làm việc trong môi trường quốc tế sau này. Các kĩ năng "mềm" khác như đi chợ, nấu ăn, giặt giũ, … cũng giúp bạn lập trình tốt hơn:)[16]

BÀI TẬP

Bt 3.8.1 Trong bài học trên, số 1,111,111,111,111,111,111 thực sự là số nguyên tố. Số này vượt quá phạm vi kiểu `int` (2^{32}) nhưng vẫn trong phạm vi kiểu `long long` (2^{64}). Hãy sửa hàm kiểm tra tính nguyên tố ở Mã 3.8.5 để cho phép kiểm tra trên số nguyên `long long` (thay vì `int`) và viết chương trình dùng hàm này kiểm tra số lớn ở trên có là số nguyên tố không. (*Gợi ý*: học hỏi Mã 3.5.2 cách làm việc với số nguyên lớn `long long`. Chạy thử với các số nguyên tố không quá lớn như 1,073,676,287 hay 68,718,952,447 hay 1,125,899,839,733,759 trước để đảm bảo là bạn sửa đúng. Với số lớn trên có thể bạn sẽ phải đợi … đợi …)

Bt 3.8.2 Như là bài tập tra cứu, bạn hãy tìm hiểu về số nguyên tố và các thuật toán kiểm tra tính nguyên tố. Bạn có cài đặt được một thuật toán kiểm tra tính nguyên tố nào đó tốt hơn thuật toán ta đã dùng trong bài học trên không? Kiểm lại (thời gian chạy) với số lớn trong Bài tập 3.8.1. (*Gợi ý*: có thể xuất phát từ bài viết "Prime number" trên trang Wikipedia[17].)

Bt 3.8.3 Bài toán trăm trâu: "Trăm trâu ăn trăm bó cỏ/Trâu đứng ăn 5/Trâu nằm ăn 3/Trâu già 3 con ăn 1 bó/Hỏi có bao nhiêu trâu mỗi loại?". Hãy viết chương trình giải bài toán trên. Bạn có biết giải theo cách Toán của bài toán này không?

---

[16] Không đùa đâu, xem lại Bài 3.1 sẽ rõ.

[17] https://en.wikipedia.org/wiki/Prime_number

Bt 3.8.4 Viết chương trình giải phương trình: $ax^3 + bx^2 + cx + d = 0$. (*Gợi ý*: tham khảo tài liệu như Wikipedia tiếng Việt – Phương trình bậc ba.)

Bt 3.8.5 Xem lại cách tính căn bậc 2 của một số thực dương ở Bài tập 1.5.9. Bạn có biết tại sao nó được gọi là *phương pháp Newton* không? Hãy tra cứu để tìm hiểu về phương pháp Newton[18] và cài đặt phương pháp này để tính căn bậc 3 của một số thực dương.

Bt 3.8.6 Làm tương tự Bài tập 3.8.5 với *phương pháp Secant* (Secant method).

---

[18] Có thể xuất phát từ https://en.wikipedia.org/wiki/Newton%27s_method. Lưu ý là bạn cần có một ít kiến thức Toán về đạo hàm. Bạn cũng không cần (và không nên) hiểu cặn kẽ, chủ yếu coi phần mô tả sơ lược, các ví dụ minh họa, hướng dẫn cài đặt (mã giả hay mã thật). Các phần khác (khó hơn) chừa lại cho "những người có quan tâm":) Lưu ý, như vậy, kĩ năng tra cứu, tìm hiểu bao gồm cả kĩ năng tiếp thu chọn lọc cái cần thiết, phù hợp và bỏ qua những cái không/chưa cần thiết hay không phù hợp nhé!

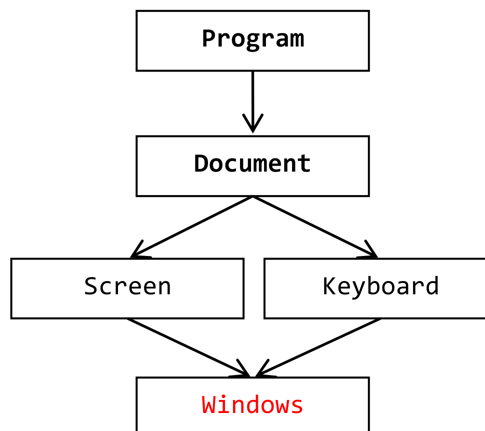
BÀI 3.9

Trong phần Mở rộng 1.7 ta đã biết về ASCII Art. Đó là nghệ thuật "vẽ bằng chữ". Trong bài này ta sẽ viết một chương trình cho phép người dùng sáng tác những "bức tranh chữ" như vậy. Chương trình này khá phức tạp với nhiều module và là dạng phần mềm thương mại (chứ không phải dạng chương trình đơn giản giải các bài toán hàn lâm) cho nên đây cũng là dịp để ta thực hành Bài 3.7 và Bài 3.8.

Như mọi chương trình bước đầu tiên là ta *xác định bài toán*. Đối với các phần mềm thì là *xác định yêu cầu của phần mềm*. Phần mềm của ta hoạt động giống như phần mềm MS Paint và MS Word. Phần mềm của ta có một bảng vẽ hình chữ nhật để người dùng vẽ trên đó (giống MS Paint hay các phần mềm đồ họa khác). Tuy nhiên ta sẽ cho người dùng gõ các kí tự trên đó (giống MS Word). Về mặt kĩ thuật ta không cho người dùng dùng chuột (tức là dùng chuột để chọn điểm vẽ) mà người dùng tương tác hoàn toàn qua bàn phím (chẳng hạn ta có con chạy (cursor) cho biết vị trí kí tự hiện hành và các phím mũi tên giúp di chuyển con chạy này, khi người dùng gõ kí tự thì kí tự sẽ được vẽ ngay tại vị trí con chạy). Về kết xuất ta cũng sẽ cho "dùng màu" để bức tranh sinh động hơn. Ta sẽ gọi phần mềm này là Char Paint.

Bước *phân tích và thiết kế* giúp ta đưa ra sơ đồ module của chương trình như sau. Sơ đồ này giống như sơ đồ phụ thuộc hàm ở Bài 3.7 nhưng ở đây là *sơ đồ phụ thuộc module* (các nút là các module).

Hình 3.9.1 – Sơ đồ module của chương trình



Đầu tiên, chương trình của ta có dùng các đặc trưng cấp thấp của hệ điều hành đó là bàn phím và màn hình (cửa sổ Console). Những đặt trưng này không có trong thư viện chuẩn (ta cũng đã nhập từ bàn phím và xuất ra màn hình tuy nhiên rất hạn chế, ở đây ta cần linh hoạt hơn chẳng hạn ta cần nhận diện các phím mũi tên hay chức năng (F1, F2, …), ta cũng sẽ xuất ra một vị trí nào đó hay cho phép màu với các kí tự, những đặc trưng này vượt quá mức thư viện chuẩn). Ở đây ta cũng dùng thư viện nhưng ở mức thấp hơn tức là thư viện của hệ điều hành (môi trường thực thi). Mà khi đó dĩ nhiên ta phải nhắm đến việc ứng dụng của ta sẽ chạy trên hệ điều hành này. Ở đây tôi chọn làm trên hệ điều hành Windows nên sẽ dùng thư viện của hệ điều hành Windows. Tuy nhiên bằng cách dùng thư viện phù hợp của hệ điều hành tương ứng mà bạn có thể port (chuyển) chương trình này dễ dàng sang các hệ điều hành khác. Để làm điều này ta đã cố ý cô lập sự phụ thuộc hệ điều hành vào các module cụ thể (và vào càng ít module càng tốt). Ở sơ đồ trên, module Windows được tô màu đỏ (và để ở dưới cùng) để nói đây không phải là module ta viết, đó là module có sẵn của hệ điều hành mà ta sẽ dùng. Module Screen quản lý việc xuất ra màn hình và module Keyboard quản lý bàn phím. Module Document là module trọng tâm của chương trình (viết nặng nhất, quan trọng nhất, quản lý, tương tác với các module khác, …) nó giúp quản lý một document ("bảng vẽ chữ"). Như sơ đồ cho thấy Document dùng Screen cho việc xuất ra màn hình và dùng Keyboard để nhận tương tác bàn phím từ người dùng. Module Program thì cực kì đơn giản nó chỉ chứa hàm main quản lý logic chương trình, còn logic nghiệp vụ sẽ do Document đảm trách.

Từ sơ đồ ta thấy chỉ có module Screen và Keyboard là phụ thuộc hệ điều hành (cấp thấp), như vậy nếu ta muốn chuyển chương trình này sang các hệ điều hành khác (như Linux, OS) thì ta chỉ cần sửa cài đặt của hai module này (mà không sửa gì các module khác). Mà điều đó được làm đơn giản bằng cách thay vì dùng module Windows thì ta dùng module tương ứng với hệ điều hành khác.

Toàn bộ mã nguồn của chương trình được cho kèm với tài liệu này. Bạn nên dùng mã nguồn đó, chạy thử và theo những phân tích dưới đây để nắm và tự chỉnh sửa, mở rộng, hoàn chỉnh theo yêu cầu hay ý tưởng của mình để rèn luyện kĩ năng lập trình nhé.

Module Keyboard quản lý bàn phím cấp thấp. Ta nói là cấp thấp là vì dưới mức thư viện chuẩn, nghĩa là thư viện chuẩn không cung cấp các chức năng này. Thư viện chuẩn chỉ nhập kí tự, ở đây ta nhập phím. Các phím như mũi tên trái, phải, … hay các phím F1, F2, … không là phím tạo kí tự. Để làm được điều này Keyboard sẽ dùng các chức năng (hàm) từ thư viện của hệ điều hành, ở đây là các hàm trong `conio.h`. Đây là file tiêu đề khai báo các hàm (không chuẩn) dùng chủ yếu ở các hệ điều hành hỗ trợ MS-DOS mà

Windows là hệ điều hành hỗ trợ đầy đủ nhất. Bạn tra cứu để hiểu hơn nhé[1]. Ở đây ta dùng 2 hàm là: kbhit để xác định người dùng có nhấn phím nào đó không và hàm getch để lấy mã kí tự (đúng hơn là mã phím) mà người dùng đã nhấn. Lưu ý hàm getch không hiển thị kí tự (nếu là phím ứng với kí tự) ra màn hình. Ở đây ta tách bạch việc nhận kí tự và xuất kí tự để mang lại sự linh hoạt cao nhất. Nhận phím (Keyboard đảm trách), xử lý (Document đảm trách) và xuất (Screen đảm trách). Như đã nói trong Bài 4.7, giao diện của Keyboard được để trong file tiêu đề keyboard.h như sau.

Mã 3.9.1 – File tiêu đề keyboard.h (của module Keyboard)

```
#define KEY_BACKSPACE   0x108
// ...

int keypressed();
int getkey();
int isnormchar(int key);
```

Keyboard cung cấp 3 hàm cho bên ngoài là: hàm keypressed cho biết có phím được nhấn không, hàm getkey cho biết mã của phím đã được nhấn và hàm insnormchar giúp kiểm tra một phím có phải là phím tạo kí tự thông thường không. Ngoài ra file tiêu đề cũng khai báo các hằng tượng trưng cho mã các phím. Mã phím là một con số nguyên nhưng ta nên dùng hằng tượng trưng để gợi nhớ đến phím tương ứng. Chẳng hạn mã của phím Backspace (phím xóa lùi ←) là 0x108 nhưng 0x108 là con số khó nhớ và có thể thay đổi (con số này có thể nói là nội bộ của Keyboard và cài đặt của Keyboard có thể quyết định dùng con số khác[2]) nhưng KEY_BACKSPACE thì gợi nhớ và sẽ cố định (file tiêu đề luôn giữ hằng tượng trưng này mặc dù con số cụ thể, 0x108, có thể thay đổi). Như vậy bên ngoài nên dùng hằng tượng trưng để định danh phím Backspace chứ không nên dùng con số 0x108[3].

Cài đặt của hàm keypressed (trong file keyboard.cpp) như sau.

Mã 3.9.2 – Cài đặt hàm keypressed (của module Keyboard)

```
#include <conio.h>
int keypressed()
{
    return kbhit();
}
```

[1] Trước hết tìm hiểu conio.h trong Wikipedia (search: wiki + conio.h).

[2] Như bạn sẽ thấy, con số 0x108 không phải là lựa chọn tất yếu mà đơn giản là một lựa chọn hợp lý, có thể thay đổi.

[3] Điều này cũng tương tự danh bạ điện thoại: tên liên lạc được dùng thay cho số điện thoại.

Hàm `kbhit` cung cấp đầy đủ chức năng mà `keypressed` cần nên cài đặt của `keypressed` đơn giản là lời gọi đến `kbhit` (và trả về cùng kết quả). Về mặt mã thì việc có một hàm mà công việc chỉ là gọi hàm khác thì khá ngớ ngẩn: tại sao ai đó dùng keypressed mà không dùng trực tiếp kbhit (sẽ nhanh hơn). Bạn có thể liên tưởng `keypressed` như là "cò dịch vụ" của `kbhit`. Tuy nhiên vấn đề là trên các hệ điều hành khác nhau thì công việc kiểm tra phím được nhấn được cung cấp bằng các cách khác nhau (không phải luôn là hàm `kbhit`, thậm chí không phải là file tiêu đề `conio.h`). Ta đã cô lập các thay đổi cấp thấp này vào Keyboard. Như vậy bên ngoài (chẳng hạn Document) sẽ luôn dùng `keypressed` để kiểm tra phím được nhấn mà không cần để ý đến chi tiết khác nhau trên các hệ điều hành khác nhau, việc này Keyboard sẽ đảm trách. Bạn có thể hình dung là bằng việc thay Dòng 1 bằng chỉ thị `#include` file tiêu đề phù hợp và thay thân hàm `keypressed` bằng cài đặt phù hợp (chẳng hạn thay Dòng 4 bằng lời gọi đến hàm phù hợp) thì Keyboard sẽ quản lý được bàn phím trên hệ điều hành khác và chương trình của ta có thể chạy ngon lành trên hệ điều hành khác mà không phải sửa gì khác ngoài cài đặt của Keyboard[4]. Ta nói `keypressed` là hàm bọc (wrapper) của hàm `kbhit`, hay ở mức "vĩ mô" hơn, ta nói Keyboard là module bọc chức năng bàn phím.

Cài đặt của hàm `getkey` (trong file `keyboard.cpp`) như sau.

Mã 3.9.3 – Cài đặt hàm getkey (của module Keyboard)

```
int getkey()
{
    int c = getch();
    if(c == 0xE0)
    {
        int code = getch();
        switch(code)
        {
            case 0x4b: return KEY_LEFT;
            // ...
        }
    }
    //...
}
```

Hàm `getch` (trong `conio.h`) sẽ trả về mã phím của phím được nhấn. Nếu là phím ứng với các kí tự ASCII thông thường thì mã đó là mã ASCII của kí tự tương ứng với phím. Nếu là các phím chức năng hay các phím mũi tên thì mã phím là `0` hoặc `0xE0` và lời gọi thứ hai đến `getch` sẽ trả về mã của phím đó. Chẳng hạn với phím mũi tên trái thì lời gọi thứ nhất đến `getch` sẽ

[4] Bạn có thể hình dung việc một sản phẩm bao gồm nhiều phần mà mỗi phần có thể được sản xuất bởi các nhà sản xuất khác nhau.

trả về 0 và lời gọi thứ hai đến getch sẽ trả về mã của phím mũi tên trái là 0x4b. Lưu ý là khi trả về mã phím ta nên dùng hằng tượng trưng KEY_LEFT chứ không nên dùng số cụ thể (0x24b) như Dòng 9 cho thấy. Cũng lưu ý là các thông tin cụ thể (chẳng hạn mã 0x4b của mũi tên trái) là phụ thuộc hệ điều hành. Như là một bài tập tra cứu, bạn hãy tìm nguồn tin cậy cho các thông tin này[5]. Tuy nhiên để chắc ăn hơn cũng như là một cách tra cứu, bạn hãy dùng chương trình sau để biết chính xác các mã phím trả về bởi getch.

Mã 3.9.4 – Chương trình khám phá mã phím trả về bởi getch

```
#include <stdio.h>
#include <conio.h>
int main()
{
    while(1)
        printf("%x\n", getch());
    return 0;
}
```

Bạn chạy chương trình trên, nhấn phím muốn biết mã, sẽ nhận được mã phím. Chẳng hạn nhấn phím mũi tên trái sẽ nhận được 2 dòng xuất ra là E0 và 4b.

Cài đặt của hàm isnormchar (trong file keyboard.cpp) như sau.

Mã 3.9.5 – Cài đặt hàm isnormchar (của module Keyboard)

```
int isnormchar(int key)
{
    return 0 < key && key < 256;
}
```

Lưu ý là mã phím trả về bởi hàm getkey (của Keyboard) đã được "trừu tượng hóa" và không liên quan gì đến mã phím trả về của hàm getch (của hệ điều hành). Xem các chỉ thị #define trong keyboard.h bạn thấy rằng Keyboard đã cố tình bố trí mã phím theo 4 loại phím: các phím chức năng (F1, …, F10) có mã có dạng 0x3?? (do đó lớn hơn 0x300), các phím mũi tên có mã có dạng 0x2?? (do đó nhỏ hơn 0x300 nhưng lớn hơn 0x200), các phím đặc biệt khác có mã có dạng 0x1?? (do đó nhỏ hơn 0x200 nhưng lớn hơn 0x100), các phím ứng với kí tự thông thường có mã là mã ASCII của kí tự (do đó nhỏ hơn 0x100 là 256). Như vậy rất dễ hiểu khi hàm isnormchar, kiểm tra key có phải là mã của kí tự thông thường hay không, được viết như trên. Hiển nhiên là việc chọn số nguyên cụ thể nào dùng làm mã cho phím cụ thể nào là do ta (Keyboard) hoàn toàn quyết định nhưng việc bố trí chúng cho hợp lý là điều ta nên cân nhắc[6]. Bạn có thấy hết cái hay của cách mà tôi đã

[5] Tôi cũng đã tra cứu (mã các phím) trong một hay vài trang web khá tin cậy nào đó mà tôi không nhớ nữa:)

[6] Điều này cũng giống như việc "thiết kế" bảng mã ASCII.

bố trí không? Cũng lưu ý là hiện thời Keyboard không "hỗ trợ" đầy đủ tất cả các phím. Chẳng hạn chỉ có các phím chức năng F1 tới F10 được hỗ trợ. Bạn có thể mở rộng Keyboard để nó hỗ trợ thêm các phím khác không? Điều này là không cần thiết trong chương trình CharPaint vì CharPaint không dùng nhiều phím chức năng (bạn sẽ thấy module Document cũng chỉ dùng các phím chức năng từ F1 đến F5). Tuy nhiên module Keyboard có thể dùng cho các "đối tác" khác (tôi thậm chí có thể bán nó riêng lẻ cho người khác dùng làm module quản lý bàn phím). Khi đó có lẽ nó nên cung cấp đầy đủ hơn (lấy tiền của người ta mà lị:)).

Tương tự module Keyboard giúp quản lý bàn phím thì module Screen giúp quản lý màn hình (xuất ra cửa sổ Console). Ta đã biết cách xuất ra màn hình dùng các hàm thư viện chuẩn tuy nhiên ta cần can thiệp ở mức thấp hơn, chẳng hạn duy chuyển con chạy hay xuất có màu. Như là bài tập rèn luyện kĩ năng đọc hiểu mã, tra cứu, bạn hãy tự tìm hiểu nhé. Sau đây là vài mô tả:

- Các hàm quản lý màn hình (cửa sổ Console) trên Windows được khai báo trong file tiêu đề `windows.h`[7].
- Hàm `GetStdHandle` với đối số `STD_OUTPUT_HANDLE` trả về "handle" của cửa sổ Console. Handle này giúp ta làm việc với cửa sổ Console. Các hàm làm việc với cửa sổ Console đều cần handle này. Do đó để đơn giản, ta sẽ lấy về handle này một lần duy nhất khi khởi động module, chứa nó trong biến toàn cục `_hscr`, và dùng nó trong tất cả các hàm sau đó. Đây là một trường hợp cho thấy cách dùng biến toàn cục hợp lý. Hơn nữa biến `_hscr` chỉ được dùng riêng cho module Screen nên nó được khai báo trong file cài đặt (`screen.cpp`) chứ không phải file giao diện (`screen.h`). Ở đây ta dùng qui ước đặt tên bắt đầu bằng kí tự gạch dưới (_) cho các biến (hay hàm) dùng riêng cho module (nghĩa là không public ra ngoài). Hiển nhiên là bên ngoài khi dùng module Screen phải gọi hàm `initscr` trước hết.
- Hàm `SetConsoleCursorPosition` giúp đặt con chạy tới vị trí mong muốn. Đối số đầu tiên là handle của cửa sổ Console, còn đối số thứ hai có kiểu là `COORD` chỉ ra vị trí. Chính xác thì `COORD` là kiểu cấu trúc mà ta sẽ tìm hiểu sau, còn bây giờ, đại khái là nó giúp chỉ ra cột và dòng của vị trí[8].

---

[7] Đây là file tiêu đề lớn nhất, quan trọng nhất khi dùng các hàm/dịch vụ/chức năng của Windows.

[8] Bạn phải mở chương trình lên vừa đọc mã vừa đối chiếu với những mô tả này nhé. Chẳng hạn, ở đây, bạn đang phải đọc dòng mã đầu tiên của hàm `gotoscr` đó.

- Hàm FillConsoleOutputCharacter giúp xuất kí tự có mã ASCII được cho ra màn hình tại vị trí mong muốn còn hàm WriteConsoleOutputCharacter thì xuất ra một chuỗi.
- Windows cho phép xuất kí tự với màu chữ và màu nền mong muốn ra cửa sổ Console nhưng số lượng màu rất hạn chế[9]. Hàm FillConsoleOutputAttribute giúp đặt màu chữ và màu nền cho một dãy các ô kí tự bắt đầu từ vị trí nào đó. Hàm này nhận màu chữ và màu nền "trộn chung" trong một số nguyên kiểu WORD[10] gọi là thuộc tính. Thế quái nào mà một số nguyên lại mô tả cả màu chữ lẫn màu nền?!:) Module Screen sẽ tách bạch màu chữ riêng, màu nền riêng và định nghĩa kiểu màu COLOR. Kiểu này thực ra là kiểu int (typedef int COLOR;), tức là ta dùng mã màu để mô tả màu[11]. Thật sự thì trên cửa sổ Console ta chỉ có 16 màu (bao gồm đậm và nhạt của 8 màu cơ bản). Các hằng tượng trưng COLOR_??? được #define trong screen.h cho thấy 16 màu đó. Ở đây việc bố trí mã màu như trong screen.h là có ý đồ (nếu bạn đổi các con số này thì việc cài đặt các hàm của Screen sẽ đổi theo, cụ thể là hàm _getAttribute). Hàm khó hiểu nhất của Screen chính là hàm _getAttribute. Trước hết nó là hàm riêng của Screen nên không được khai báo trong screen.h mà chỉ có cài đặt trong screen.cpp, hơn nữa theo qui ước ta đặt tên hàm bắt đầu bằng kí tự _. Hàm này giúp "trộn" màu nền (backclr) và màu chữ (textclr) vào thuộc tính att và trả về. Để làm điều này hàm sử dụng thao tác OR-bít (toán tử |) mà bạn cũng có thể đoán rằng |= là toán tử gán kép cho toán tử |. OR-bít là gì và tại sao lại OR-bít? Ta sẽ tìm hiểu sau nhé. Tạm thời bạn tra cứu để hiểu sơ sơ là được rồi:) Cũng lưu ý các hằng tượng trưng BACKGROUND_??? Và FOREGROUND_??? Là do Windows #define.

Hiển nhiên module Screen cũng có thể được cho hoặc bán riêng cho các đối tác khác dùng để quản lý việc xuất ra màn hình:)

Module Program quan trọng vì nó chứa ngõ vào chương trình (hàm main) nhưng nó là module cực kì đơn giản vì mọi công việc đều do module đảm nhiệm. Có thể nói Program đóng vai trò giao tiếp với môi trường. Môi trường thấy và thực thi chương trình CharPaint qua Program. Ta sẽ thấy trường hợp module giao tiếp này phức tạp sau, còn ở đây nó rất đơn giản, nó chỉ có duy nhất hàm main (thậm chí không có file tiêu đề cho module này):

---

[9] Trong cửa sổ GUI thì việc dùng màu (và các chức năng đồ họa khác) thì rất phong phú. Ta sẽ tìm hiểu sau.

[10] Kiểu WORD được Windows định nghĩa là: typedef unsigned short WORD;

[11] Như mã ASCII được dùng để mô tả kí tự.

Mã 3.9.6 – File cài đặt của module Program (file `program.cpp`)

```
1  #include "document.h"
2
3  int main()
4  {
5      initdoc(80, 50);
6      rundoc();
7      exitdoc();
8
9      return 0;
10 }
```

Như bạn thấy, cài đặt của hàm `main` cũng rất đơn giản: gọi hàm `initdoc`, `rundoc`, `exitdoc` từ module Document rồi kết thúc chương trình. Module Document là trọng tâm của chương trình với file tiêu đề như sau.

Mã 3.9.7 – File tiêu đề `document.h` (của module Document)

```
#define WIDTH_MAX    80
#define HEIGHT_MAX   50

void initdoc(int width, int height);
void rundoc();
void exitdoc();
```

Ngoài việc public 3 hàm trên (mà Program đã dùng) thì Document cũng public hai hằng tượng trưng là `WIDTH_MAX` và `HEIGHT_MAX`. Bảng vẽ kí tự được gọi là document[12]. Nó là khung chữ nhật gồm *m* cột và *n* dòng với mỗi ô là kí tự (cùng với màu kí tự và màu nền của ô đó). Ta gọi *m* là chiều rộng và *n* là chiều cao[13] của bảng vẽ. Hàm `initdoc` giúp tạo bảng vẽ có chiều rộng là `width` và chiều cao là `height`. Hằng tượng trưng `WIDTH_MAX` và `HEIGHT_MAX` xác định giá trị được phép lớn nhất của chiều rộng và chiều cao. Bên ngoài không được phép tạo bảng vẽ vượt quá kích thước này. Chẳng hạn Program đã tạo bảng vẽ kích thước tối đa bằng Dòng 5 Mã 3.9.6. Hiển nhiên là Document có thể sửa giá trị của hai hằng tượng trưng này để cho phép bảng vẽ kích thước lớn hơn. Tuy nhiên điều này phụ thuộc vào "kích thước vật lý" của cửa sổ Console. Hàm `rundoc` giúp người dùng "vẽ" lên bảng vẽ, nghĩa là nhận tương tác của người dùng và kết xuất tương ứng lên bảng vẽ. Tương tác ở đây đều qua bàn phím (chứ không dùng chuột hay các thiết bị nhập khác): các phím mũi tên giúp di chuyển con chạy, các phím kí tự bình thường giúp đặt kí tự vào vị trí hiện tại của con chạy, các phím xóa, các phím chức năng giúp thực hiện các thao tác đặc biệt (như bật tắt

[12] Bảng vẽ sao lại gọi là document chứ?! Mình thích thì mình gọi thôi:)

[13] Bảng vẽ 2 chiều thì có rộng và dài chứ làm gì có cao?! Cái này là tôi bắt chước gọi theo người khác còn người ta thích thì người ta gọi thôi:)

chế độ Insert, xóa bảng vẽ, đặt màu chữ, màu nền, …) và phím đóng bảng vẽ. Khi người dùng kết thúc tương tác với bảng vẽ (đóng bảng vẽ) thì hàm `rundoc` kết thúc. Hàm `exitdoc` "dọn dẹp" bảng vẽ và thông báo kết thúc.

Khác với sự đơn giản của file tiêu đề `document.h`, file cài đặt `document.cpp` lại khá phức tạp. Đó là vì Document dùng thêm nhiều hằng tượng trưng, biến toàn cục và hàm nội bộ (tất cả đều được đặt tên bắt đầu bằng kí tự _ như qui ước). Nhiệm vụ của Document là cung cấp 3 hàm public ở trên nhưng để làm được việc này nó sẽ dùng thêm nhiều hàm/biến/hằng tượng trưng nội bộ để phụ giúp. Các hàm phụ giúp này còn được gọi là các *hàm trợ thủ*. Hiển nhiên là bên ngoài không biết đến chúng mà chỉ biết đến 3 hàm public trên thôi. Về mặt kĩ thuật, các biến toàn cục và hàm nội bộ này được khai báo với từ khóa `static` để cho chúng có phạm vi file và do đó không thể truy cập từ mã trong các file khác.

Cũng nhớ rằng Document dùng Keyboard để quản lý bàn phím và Screen để quản lý màn hình (chứ không trực tiếp tương tác mức thấp bằng cách gọi hệ điều hành). Như là bài tập rèn luyện kĩ năng đọc hiểu mã, bạn hãy tự tìm hiểu nhé[14]. Như bao công việc phân tích và tìm hiểu khác, cách làm tốt là phân tích từ trên xuống từ là từ tổng thể đến chi tiết[15]. Chẳng hạn phân tích tổng thể các hàm chính của module, sau đó phân tích tổng thể từng hàm chính, sau đó phân tích các hàm trợ thủ. Với từng hàm trợ thủ thì xác định công việc của nó (what?, hộp đen) sau đó mới phân tích chi tiết mã nếu cần. Sau đây là vài mô tả:

- Tất cả các biến toàn cục trong Document như `_width`, `_height`, `_row`, `_col`, … đều dùng để lưu giữ thông tin hay trạng thái của bảng vẽ (như chiều rộng, chiều cao của bảng vẽ hay vị trí (dòng, cột) hiện tại của con chạy, …). Vì ta chỉ có duy nhất một bảng vẽ (document) trong toàn bộ chương trình nên các thông tin này có thể đặt thành biến toàn cục và các thao tác (hàm) của Document sẽ chia sẻ (dùng/thay đổi) chúng. Mô hình hoạt động này được gọi là "SDI" (Single Document Interface) để ám chỉ chương trình chỉ làm việc với một "tài liệu" (document)[16]. Ngược lại với mô hình này là MDI (Multiple Document Interface), kiến trúc cho phép chương trình làm việc với nhiều tài liệu cùng lúc[17]. Khi đó ta phải thiết kế module Document khác đi (chẳng hạn dùng kiểu cấu trúc mà ta sẽ học sau).

---

[14] Thật sự thì chương trình CharPaint có nhiều chức năng mà người dùng không khám phá hết. Nhưng, với tư cách là người tìm hiểu mã, bạn phải khám phá hết các chức năng của chương trình nhé.

[15] Giống như cách người họa sĩ vẽ một bức tranh: phác họa tổng thể rồi mới đến chi tiết từng bộ phận.

[16] Bạn có thể dùng thử chương trình Windows Notepad, … sẽ rõ.

[17] Bạn có thể dùng thử chương trình Adobe Photoshop hay Notepad++, … sẽ rõ.

- Một số thông số của Document mà ta không nên "code cứng" (nghĩa là dùng hằng số) mà nên dùng hằng tượng trưng (hay thậm chí là biến nếu Document có dự định cho phép bên ngoài thay đổi chúng). Vài thông số như vậy là kích thước tab (số cột mà phím tab sẽ dời con chạy đi) trong `_TAB_SIZE`, lề trái, lề trên của cửa sổ (frame) document trong `_LEFT_PAD`, `_TOP_PAD`.
- Hàm `initdoc` bắt đầu bằng việc gọi khởi động các module mà Document dùng, ở đây là gọi `initscr` để khởi động Screen (Keyboard không cần khởi động nên không có hàm nào như vậy). Sau đó đặt các giá trị ban đầu cho các thông tin/trạng thái của document. Chẳng hạn đặt chiều rộng, chiều cao của document từ tham số, đặt chế độ chèn (`_insert`, có dời con chạy qua khi gõ kí tự vào không) là bật (1), đặt kí tự trắng (`_blankchar`, kí tự dùng với các phím xóa Space/Backspace/Delete) là kí tự space, đặt kí tự chèn (`_inschar`, kí tự dùng với phím Enter) là kí tự `'.'`, đặt màu nền cho ô (`_backcolor`) là màu xanh sáng (`COLOR_LTBLUE`), đặt màu chữ (`_forecolor`) là màu trắng sáng (`COLOR_LTWHITE`). Sau đó gọi các hàm trợ thủ: `_drawframe` để vẽ khung quanh document, `_clear` để xóa (điền kí tự trắng với màu nền, màu chữ hiện tại vào toàn bộ document), `_moveto` để đặt vị trí con chạy là ở dòng 0 và cột 0, `_drawinfor` để vẽ thanh trạng thái (cho biết thông tin/trạng thái hiện hành của document như vị trí con chạy, …) ở trên khung, `_drawhelp` để vẽ thanh giúp đỡ (thông tin các phím tắt) ở dưới khung.
- Hàm `rundoc` là quan trọng nhất. Đó là nơi tương tác của người dùng với document diễn ra. Nó cài đặt vòng lặp tương tác[18]: người dùng tương tác (nhấn phím), document phản ứng (làm gì đó). Vòng lặp này diễn ra hoài cho đến khi người dùng đóng document (ở đây là nhấn phím Esc). Tương tác thông thường nhất của người dùng là gõ kí tự (bình thường, `isnormchar`) vào bảng vẽ tại vị trí hiện hành của con chạy và document sẽ kết xuất kí tự đó ra bảng vẽ (và dời con chạy qua phải nếu cần) bằng hàm `_writechar`. Dạng tương tác thứ hai là người dùng nhấn các phím mũi tên (trái, phải, …) và document sẽ dời con chạy đi tương ứng (qua trái, qua phải, …) bằng hàm `_move` hay `_moveto`. Dạng tương tác thứ ba là người dùng nhấn các phím xóa hay Enter và document sẽ kết xuất kí tự trắng hay kí tự chèn bằng hàm `_writechar`. Dạng tương tác thứ tư là người dùng nhấn các phím chức năng như Insert để bật/tắt chế độ chèn, F1 để "xóa trắng" bảng vẽ (và "kêu bíp" để gây chú ý bằng hàm `_beep`, mà đơn giản là dùng kí tự BEL mã 7), F2 để đặt màu chữ, F3 để đặt màu

[18] Còn được gọi là vòng lặp thông điệp (message loop).

nền, F4 để chọn kí tự trắng, F5 để chọn kí tự chèn. Dạng tương tác cuối cùng là người dùng đóng document bằng phím Esc. Khi đó hàm `rundoc` kết thúc.

- Hàm `exitdoc` "dọn dẹp" document (thường thì đó là giải phóng các tài nguyên, … nhưng ở đây thì không có gì để dọn dẹp cả). Hàm này đưa ra thông điệp lưu luyến "Bye!" rồi đợi người dùng nhấn phím (bất kì) rồi kết thúc (mà không quên kêu bíp:)).
- Tương tác người dùng trong việc chọn màu (chữ/nền) và chọn kí tự (trắng/chèn) thật sự phức tạp. Đối với các ứng dụng đồ họa (ứng dụng lớn thực tế) thì phải dùng hộp thoại chuyên biệt như hộp thoại chọn màu[19], chọn kí tự. Khi đó tương tác người dùng (chọn màu, chọn kí tự) sẽ diễn ra trên hộp thoại đó. Ở đây hàm `_choosecolor` và `_choosechar` sẽ "hiển thị" một hộp thoại như vậy. Khi `_choosecolor` được gọi, tương tác người dùng sẽ tạm thời chuyển qua thành chọn màu (chứ không tương tác trên bảng vẽ nữa). Do đó `_choosecolor` cũng có một vòng lặp tương tác mà trong đó người dùng nhấn phím Up/Down để duyệt qua các màu được phép (trong số 16 màu). Ở đây logic duyệt dựa trên cách bố trí màu "liên tục" của Screen với màu đầu là `COLOR_BLACK` và màu cuối là `COLOR_LTWHITE`. Hơn nữa khi người dùng duyệt màu thì "hộp thoại" chọn màu sẽ "hiển thị" màu đang chọn cho người dùng biết. Để kết thúc tương tác chọn màu (và như vậy sẽ "đóng" hộp thoại chọn màu), người dùng nhấn phím Enter hoặc Esc. Phím Enter nghĩa là muốn chọn màu mới đó (tương ứng với "nút OK" của hộp thoại) và như vậy `_choosecolor` sẽ đặt giá trị màu mới cho màu chữ (`_forecolor`) hay màu nền (`_backcolor`). Phím Esc nghĩa là muốn hủy thao tác chọn màu, giữ lại màu cũ (tương ứng với "nút Cancel" của hộp thoại) và như vậy `_choosecolor` không cần làm gì cả (dĩ nhiên là cũng kết thúc vòng lặp). Hộp thoại chọn kí tự (hàm `_choosechar`) thì còn phức tạp hơn nữa. Ngoài cách duyệt qua các kí tự được phép (theo thông số `_STARTCHAR` và `_ENDCHAR` thì đó là tất cả 256 kí tự ASCII) bằng các phím Up/Down/PgUp/… thì người dùng có thể nhập trực tiếp kí tự hoặc nhập mã ASCII (bằng phím \ và sau đó là 2 chữ số cơ số 16 của mã ASCII). Bạn đọc mã để hiểu nhé[20].

Bạn đã học được nhiều điều trong bài này. Điều quan trọng nhất là về kiến trúc (cách thiết kế) của phần mềm. Các quyết định lựa chọn việc phân

---

[19] Bạn có thể dùng thử chương trình Windows Paint (hay các phần mềm đồ họa khác như Adobe Photoshop) để thấy một hộp thoại chọn màu như vậy.

[20] Nếu bạn đọc mã mà hiểu được thì chứng tỏ tôi viết mã hay còn như bạn đọc mà không hiểu thì chứng tỏ bạn đọc mã dở:)

chia module, lập sơ đồ module, phân chia các hàm riêng, công khai, truyền nhận tham số giữa các hàm, biến toàn cục, … được gọi chung là *kiến trúc của phần mềm*. Nó là phần quan trọng nhất của một phần mềm[21]. Một phần mềm có kiến trúc càng đơn giản, rõ ràng thì càng bền vững, càng ít lỗi và càng dễ dàng chỉnh sửa, mở rộng. Ngoài ra, bạn cũng học được cách chia phần mềm ra các module độc lập, phụ thuộc nhau rất ít qua các giao tiếp và đặc biệt cần cô lập các module cấp thấp (phụ thuộc phần cứng, kiến trúc máy, công nghệ, hệ điều hành, nền tảng, môi trường...). Điều này giúp tạo nên các module có thể dùng lại cho các mục đích khác và có thể được phát triển bởi những đối tác (lập trình viên, nhóm, công ty, …) khác nhau. Đây cũng là cách mà một phần mềm được xây dựng và phát triển thực tế trong thời đại toàn cầu như ngày nay.

BÀI TẬP

Bt 3.9.1 Bảng mã 437 có nhiều kí tự đồ họa (xem lại Bài 1.7). Hãy vận dụng các kí tự này để cung cấp thêm các chức năng vẽ hình (với màu sắc) cho chương trình trên (hình chữ nhật với các biên đơn, biên kép, các khối hình khác, …). (*Gợi ý*: hãy sáng tạo.)

Bt 3.9.2 Thêm chức năng vẽ các chữ cái đẹp (chữ to, nhiều nét) với các mẫu chữ (font) khác nhau (xem Bài 2.7).

Bt 3.9.3 Hãy tìm hiểu, tra cứu một số template "hình chữ" và thêm chức năng chèn các hình này. (Chẳng hạn các emoticon[22].)

Bt 3.9.4 (Mở rộng[23]) Xem lại phần Mở rộng 1.7 về cách kết xuất chữ cái tiếng Việt có dấu. Thêm chức năng "gõ" chữ cái tiếng Việt có dấu bằng cách cung cấp một "bàn phím ảo tiếng Việt".

---

[21] Tương tự như kiến trúc của các công trình xây dựng.

[22] https://en.wikipedia.org/wiki/List_of_emoticons

[23] Yêu cầu của các bài tập được đánh dấu "Mở rộng" là khó nhưng chúng xứng đáng để bạn bỏ công làm: bạn sẽ được một chương trình rất thú vị và công lực lập trình của bạn cũng tăng nhanh đáng kể.

BÀI 4.1

Chương trình sau đây tìm số lớn nhất trong dãy *n* số nguyên mà người dùng nhập vào. Bạn hãy gõ và chạy thử.

Mã 4.1.1 – Chương trình tìm số lớn nhất trong *n* số người dùng nhập

```
1  #include <stdio.h>
2  #include <limits.h>
3  int main()
4  {
5      int n;
6      printf("Ban muon nhap bao nhieu so: ");
7      scanf("%d", &n);
8  
9      int x, M = INT_MIN;
10     for(int i = 1; i <= n; i++)
11     {
12         printf("Nhap so thu %d: ", i);
13         scanf("%d", &x);
14         if(x > M)
15             M = x;
16     }
17     printf("So lon nhat la: %d\n", M);
18 
19     return 0;
20 }
```

Biến n chứa số lượng số mà người dùng muốn nhập. Sau khi cho người dùng nhập n ta cho người dùng lần lượt nhập n số nguyên bằng vòng lặp `for` với biến lặp `i` chạy từ 1 đến n. Biến x chứa lần lượt các số nguyên người dùng sẽ nhập. Giá trị của x như vậy sẽ phụ thuộc vào lần lặp `i` nên ta sẽ kí hiệu nó là x_i. Như vậy người dùng sẽ nhập lần lượt *n* số nguyên: x_1, x_2, ..., x_n. Trước khi thực hiện vòng lặp, biến `M` chứa số nguyên kiểu `int` nhỏ nhất được `#define` cho hằng tượng trưng `INT_MIN` trong `limits.h`[1]. Khi vòng lặp `for` thực thi lần thứ nhất (`i` = 1) thì người dùng nhập giá trị vào x (mà ta kí hiệu là x_1). Lệnh `if` ở Dòng 14 giúp đảm bảo `M` chứa giá trị của x_1. Thật vậy nếu x_1 không là `INT_MIN` thì x_1 > `M` (vì `M` đang chứa `INT_MIN` và `INT_MIN` là số `int` nhỏ nhất) nên lệnh gán `M` = x_1 được thực thi, còn nếu x_1 là `INT_MIN` thì `M` vẫn

[1] Đó là số -2,147,483,648 (-2^{31}) như trong Bài 3.5. Tuy nhiên ta nên dùng hằng tượng trưng `INT_MIN` vì kích thước kiểu `int` (và do đó giá trị nhỏ nhất này) phụ thuộc vào lựa chọn của compiler.

giữ giá trị ban đầu là `INT_MIN`. Ta cũng kí hiệu M_k là giá trị của M sau khi vòng lặp `for` thực thi xong lần `i` = k vì giá trị của `M` cũng phụ thuộc vào lần lặp. Như vậy $M_1 = x_1$. Khi vòng lặp `for` thực thi lần thứ hai (`i` = 2) thì người dùng nhập vào giá trị x (mà ta kí hiệu là x_2). Sau khi thực thi lệnh `if` thì ta có $M_2 = \max\{M_1, x_2\} = \max\{x_1, x_2\}$[2], … khi vòng lặp for thực thi xong lần `i` = k thì $M_k = \max\{M_{k-1}, x_k\} = \max\{\max\{x_1, …, x_{k-1}\}, x_k\} = \max\{x_1, …, x_k\}$ tức là biến `M` chứa số lớn nhất trong các số $x_1,…, x_k$. Do đó khi vòng lặp kết thúc (xong `i` = `n`) thì `M` chính là số lớn nhất trong `n` số đã nhập. Ở đây không có gì cao siêu, ta đã vận dụng một định lý "hiển nhiên": $\max\{x_1, …, x_k\} = \max\{\max\{x_1, …, x_{k-1}\}, x_k\}$. Tuy nhiên về mặt kĩ thuật (tức là cài đặt) thì cần để ý học hỏi trường hợp "*khởi động*" nhé (tức là việc gán giá trị đầu tiên cho `M`, `M = INT_MIN`).

Thế bây chừ ta muốn tìm số lớn thứ hai thì phải làm răng? Giả sử n số nhập vào đều khác nhau. Ta có thể dùng lại chương trình trên với `M` không phải là số lớn nhất mà là số lớn thứ hai. Nghĩa là khi `for` thực hiện xong lần `i` = k-1 thì M_{k-1} là số lớn thứ 2 trong các số x_1, …, x_{k-1}. Khi `i` = k ta cần phải tính được M_k từ x_k và M_{k-1}. Lưu ý là ta đã không trữ lại các số trước đó nên ta không "nhớ" x_1, …, x_{k-1} là số nào nên ta không làm được việc này vì từ x_k và M_{k-1} ta không đủ thông tin (hay dữ liệu) để tính M_k. Chẳng hạn nếu $x_k < M_{k-1}$ thì $M_k = M_{k-1}$ nhưng nếu $x_k > M_{k-1}$ thì M_k có thể bằng x_k hoặc bằng số lớn nhất trong các số $x_1,…, x_{k-1}$, tùy theo số nào nhỏ hơn. Từ đây ta thấy, mặc dù cần tìm số lớn thứ hai nhưng ta cũng cần lưu lại số lớn nhất. Do đó ta sửa đoạn chương trình trên lại bằng cách dùng 2 biến `M1`, `M2` cho số lớn nhất và lớn thứ hai.

Mã 4.1.2 – Đoạn chương trình tìm số lớn thứ hai

```
M1 = M2 = INT_MIN;
for(int i = 1; i <= n; i++)
{
    printf("Nhap so thu %d: ", i);
    scanf("%d", &x);
    if(x > M1)
    {
        M2 = M1;
        M1 = x;
    }
    else if(x > M2)
        M2 = x;
}
printf("So lon thu hai la: %d\n", M2);
```

Khi vòng `for` thực hiện xong lần `i` = k-1 thì $M1_{k-1}$, $M2_{k-1}$ là số lớn thứ nhất và thứ hai trong các số x_1, …, x_{k-1}. Khi `i` = k ta cần phải tính được $M1_k$, $M2_k$ từ x_k và $M1_{k-1}$, $M2_{k-1}$. Có 3 trường hợp xảy ra:

[2] Bạn tự phân tích nhé.

1) $x_k > M1_{k-1}$: $M1_k = x_k$ và $M2_k = M1_{k-1}$.
2) $M1_{k-1} > x_k > M2_{k-1}$: $M1_k = M1_{k-1}$ và $M2_k = x_k$.
3) $M2_{k-1} > x_k$: $M1_k = M1_{k-1}$ và $M2_k = M2_{k-2}$.

Hãy xem 3 trường hợp này được cài đặt bằng lệnh `if` trên như thế nào:

- Trường hợp 1: cần lưu ý phải viết `M2 = M1` trước `M1 = x`. Vì nếu không, sau lệnh: `M1 = x` (tức là $M1_k = x_k$) thì lệnh `M2 = M1` sẽ đặt $M2_k = M1_k$ chứ không phải $M1_{k-1}$.
- Trường hợp 2: ta không cần lệnh cho $M1_k = M1_{k-1}$, đơn giản là `M1` mới (ứng với k) vẫn giữ giá trị cũ (ứng với k-1).
- Trường hợp 3: ta không cần làm gì do `M1`, `M2` đều giữ các giá trị cũ nên ta, thậm chí, không viết ra trường hợp này trong code.

Bây giờ giả sử ta muốn tìm số lớn thứ ba (hay thứ tư, …) thì sao? Có lẽ bạn đã phát hiện ra khuôn mẫu. Dùng 3 (hay 4, …) biến `M1`, `M2`, `M3`, … và chia ra 4 (hay 5, …) trường hợp để tính $M1_k$, $M2_k$, $M3_k$, … từ x_k và $M1_{k-1}$, $M2_{k-1}$, $M3_{k-1}$, … Bạn hãy làm thử cho trường hợp số lớn thứ ba và thứ tư nhé. Tuy nhiên bạn đủ độ chịu khó (hay độ "trâu bò") để làm vậy không? Thậm chí nếu ta muốn tìm số lớn thứ m mà m chưa xác định trước (chẳng hạn người dùng sẽ nhập m) thì cách làm trên không dùng được.

Tại sao cách làm trên lại khó khăn đến vậy (với m lớn hay chưa xác định trước). Mấu chốt là ở chỗ, mỗi lần tính toán hay kiểm tra, ta đã chỉ dùng x_k mà không dùng các số trước đó x_1, …, x_{k-1}. Thậm chí ta có thể dùng cả n số x_1, …, x_n cơ mà ?! Vấn đề chính là ta đã không lưu trữ lại các số. Nếu "được phép" ta hãy nhận cả n số trước, *lưu trữ lại rồi sau đó mới xử lý*. Cách làm này gọi là *ngoại tuyến* (*offline*) trong khi cách đầu tiên chúng ta chỉ dùng rất ít dữ liệu hiện tại để xử lý được gọi là *trực tuyến* (*online*). Đúng hơn thì nên gọi là *online/offline đầu vào*[3]. Giả sử bạn cần xử lý n mẫu dữ liệu *tuần tự* (bạn nhận được lần lượt mẫu thứ 1, thứ 2, …, cho đến mẫu cuối cùng là mẫu thứ n[4]), chẳng hạn ở trên người dùng nhập lần lượt n số:

- *Xử lý online*: bạn nhận mẫu dữ liệu nào là xử lý liền và không lưu trữ (hay lưu trữ rất ít) các mẫu dữ liệu trước đó. Cách này có lợi là bạn không tốn (hoặc tốn rất ít) bộ nhớ lưu trữ và có ngay kết quả khi nhận xong mẫu dữ liệu cuối cùng. Hơn nữa nó có tính tương tác hay tức thời vì có các kết quả xử lý trung gian. Chẳng hạn ta có `M` là số lớn nhất từ số đầu cho đến số vừa nhập xong.
- *Xử lý offline*: bạn nhận hết dữ liệu, lưu trữ lại rồi mới xử lý toàn bộ. Cách này có nhược điểm là tốn bộ nhớ lưu trữ, không có tính tương

[3] Tôi không thấy có thuật ngữ nào như vậy. Đây là thuật ngữ tôi bịa ra nhưng chắc là có ai đó đã bịa ra trước:)

[4] Mô hình này thường xuất hiện khi dữ liệu truyền trên mạng nên chữ offline/online là thuật ngữ mạng.

tác và có độ trễ do phải đợi xử lý toàn bộ. Bù lại cách này thường dễ hơn do ta có đầy đủ dữ liệu khi xử lý.

Trở lại bài toán trên, chúng ta sẽ dùng cách xử lý offline: lưu trữ lại tất cả *n* số mà người dùng nhập sau đó xử lý rồi cho kết quả. Khi đã có đủ *n* số chúng ta có thể *sắp xếp* lại *n* số theo thứ tự giảm dần. Khi đó số lớn nhất là số đầu tiên (sau khi xắp lại) và số lớn thứ *m* là số ở vị trí thứ *m*. Làm sau có thể lưu trữ lại *n* số nguyên và thực hiện việc sắp xếp giảm dần trên *n* số đó. Ta dùng *mảng* như sau.

Mã 4.1.3 – Chương trình tìm số lớn thứ *m* bằng cách xử lý offline

```
#include <stdio.h>
void nhap(int a[], int n)
{
    for(int i = 1; i <= n; i++)
    {
        printf("Nhap so thu %d: ", i);
        scanf("%d", &a[i - 1]);
    }
}
void sapxep(int a[], int n)
{
    for(int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
            if(a[i] < a[j])
            {
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
int main()
{
    int n, x[1000];
    printf("Ban muon nhap bao nhieu so: ");
    scanf("%d", &n);

    nhap(x, n);
    sapxep(x, n);

    int m;
    printf("Ban muon tim so lon thu may: ");
    scanf("%d", &m);
    printf("So lon thu %d la: %d\n", m, x[m - 1]);

    return 0;
}
```

Nếu muốn tìm số lớn nhất bạn nhập `m` là 1. Nếu muốn tìm số lớn thứ hai bạn nhập `m` là 2. Nếu muốn tìm số nhỏ nhất (số lớn thứ cuối cùng) bạn nhập `m` là n. Cách xử lý offline này còn có ưu điểm là tổng quát (cho `m` bất kì thay vì số cố định như trong cách xử lý online).

Trong chương trình trên ta đã dùng một *mảng* x các số nguyên, cho người dùng nhập dữ liệu vào x, xử lý trên x (sắp xếp x) và truy xuất dữ liệu trong x để xuất ra kết quả. Sau đây là những điều cơ bản nhất về mảng trong C:

- Mảng giúp ta chứa dữ liệu như biến nhưng mảng cho phép ta *chứa nhiều dữ liệu* (*cùng kiểu*) thay vì chỉ một dữ liệu như biến. Chẳng hạn mảng x ở trên cho phép chứa nhiều số nguyên thay vì chỉ một số nguyên như biến n. Mỗi dữ liệu trong mảng được gọi là *phần tử* của mảng. Như vậy mỗi số nguyên trong mảng x được gọi là một phần tử của x.
- Khi khai báo mảng ta dùng cú pháp:

  <kiểu của phần tử> <tên mảng>[<số lượng tối đa các phần tử>]

  Chẳng hạn khai báo mảng x ở trên (Dòng 23):

  ```
  int x[1000];
  ```

  cho biết x là mảng chứa tối đa 1000 số nguyên. Mỗi phần tử của x là một số nguyên. Ta còn nói x là mảng các số nguyên hay mảng `int`[5].
- Không có thông tin ngầm định nào đi kèm với mảng để cho biết số lượng phần tử thực sự đang chứa trong mảng. Ta cần dùng thêm biến để chứa con số này. Số lượng thực sự các phần tử đang chứa trong mảng còn được gọi là *chiều dài* mảng. Như vậy ta cần dùng thêm biến để chứa chiều dài mảng. Vì chiều dài mảng là một con số nguyên không âm nên ta thường dùng biến kiểu `int` để chứa nó. Ở trên thì biến `n` (kiểu `int`) được dùng để chứa chiều dài của mảng x. Vì x chứa `n` số nguyên nên `n` cũng chính là số lượng số mà người dùng muốn nhập. Ta cũng cần phân biệt chiều dài mảng (là số lượng phần tử thực sự đang chứa trong mảng) với *sức chứa* của mảng (số lượng tối đa phần tử được chứa trong mảng). Hiển nhiên là chiều dài mảng có thể thay đổi (tùy theo mảng chứa thêm hay bớt đi phần tử) nhưng luôn luôn không được quá sức chứa của mảng. Ở trên thì chiều dài mảng là n không được quá sức chứa mảng là 1000. Như vậy người dùng không được phép nhập quá 1000 số[6]. Chắc chắn là bạn sẽ thắc mắc nếu ta để sức chứa là 1000 mà người dùng chỉ

[5] Ta đã gặp mảng ngay trong Bài 1.2, nhưng đó là mảng các kí tự (mảng `char`) giúp chứa nhiều kí tự, tức là chuỗi.

[6] Nếu có thể người dùng nhập nhiều hơn thì bạn cần nâng sức chứa này lên nhé, chẳng hạn 10000.

nhập có 10 số thì như vậy chẳng phải quá lãng phí bộ nhớ sao?[7] Đúng vậy. Đây là hạn chế của mảng loại này, còn gọi là *mảng tĩnh*. Các kĩ thuật cấp cao hơn như *mảng động* sẽ được luyện sau nhé.

- Mảng cũng có thể là tham số cho hàm, nhờ đó ta có thể có hàm xử lý mảng (hàm nhận mảng và thao tác trên mảng). Tham số là mảng cũng được khai báo với cú pháp:

  <kiểu của phần tử> <tên tham số mảng>[ ]

  Lưu ý là trong tham số mảng ta không có nêu sức chứa của mảng như là trong khai báo mảng. Tại sao? Nói nôm na, trong khai báo mảng ta cần cho biết sức chứa để C cấp phát vùng nhớ cho mảng (để có thể chứa được tối đa từng đó phần tử) nhưng trong tham số thì ta không cần vì C chỉ cần biết tham số đó là mảng (mà không cấp phát)[8]. Khi hàm nhận tham số mảng thì ta cũng thường có tham số đi kèm để chứa chiều dài của mảng. Ở trên thì hàm `nhap` và `sapxep` sẽ thao tác trên mảng các số nguyên nên nó sẽ có tham số đầu là mảng, ngoài ra có thêm tham số (kiểu `int`) cho biết chiều dài của mảng. Bạn xem prototype của hàm `nhap` và `sapxep` (Dòng 2 và Dòng 10). Cũng nhớ rằng tên tham số trong hai hàm không liên quan nhau, nghĩa là khác nhau cũng được. Tuy nhiên vì ta thường đặt tên cho mảng là `a`[9] và chiều dài mảng là `n` nên tình cờ hai tham số của hai hàm trên trùng tên nhau. Hơn nữa tham số mảng `a`, tham số `n` và mảng `x`, biến `n` trong `main` không liên quan gì nhau (ngoại trừ mảng `x` và giá trị của biến `n` sẽ được gán tương ứng vào tham số `a` và tham số `n` khi hàm tương ứng được gọi chạy, nếu chưa rõ thì coi lại Bài 3.3 nhé).

- Để quản lý các phần tử trong mảng thì C dùng *số hiệu* chứ không dùng tên. Điều này cũng giống như khi nhà có con đông thì thường "đặt tên bằng số" là Thằng Hai, Con Ba, Thằng Bốn, ..., Thằng Mười, Con Út:)[10]. Cách dùng số hiệu có nhiều cái lợi nhất là khi mảng có quá nhiều phần tử[11]. Số hiệu của phần tử còn được gọi là *chỉ số* của phần tử. Đó là con số nguyên không âm và trong C nó *bắt đầu từ* 0. Như vậy phần tử đầu tiên trong mảng gọi là phần tử 0 của mảng, phần tử thứ hai là phần tử 1 của mảng, ... phần tử cuối cùng trong

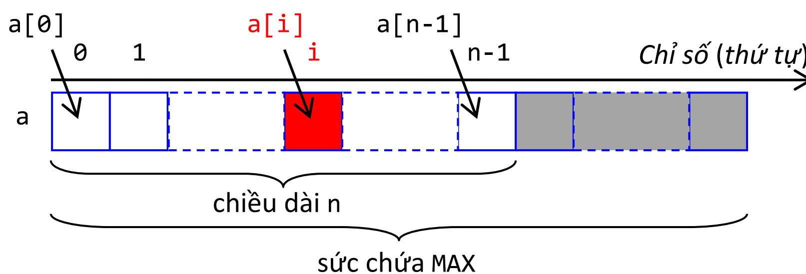
[7] Tưởng tượng ta xây cái nhà có 1000 phòng mà chỉ dùng 10 phòng.
[8] Chi tiết ta sẽ rõ sau.
[9] Viết tắt của array.
[10] Con "Mười Ráng" đó:)
[11] Ở đây ta cũng cần nhớ về bài học dùng số (nhất là số nguyên) để mô tả, biểu diễn nhiều thứ (như kí tự) một cách hiệu quả.

mảng có chiều dài *n* là phần tử có chỉ số *n* – 1[12]. Chính vì lí do này mà ta thường gọi mảng là *danh sách*.
Để truy cập phần tử trong mảng ta dùng cú pháp:

<tên mảng>[<chỉ số phần tử cần truy cập>]

Khi ta truy cập tới phần tử trong mảng thì nó tương đương như là một biến (mảng thì chứa nhiều dữ liệu nhưng phần tử thì chỉ chứa một dữ liệu như biến). Chẳng hạn ở trên sau khi sắp xếp mảng thì ta có phần tử thứ (*m* – 1) (tức là phần tử có chỉ số (*m* – 1)) chính là số lớn thứ *m* trong mảng. Do đó Dòng 33 xuất ra giá trị mong đợi (ở đây ta dùng phần tử `x[m - 1]` như là một r-value). Cũng lưu ý là trong ngôn ngữ đời thường ta thường đánh số từ 1, tức là thứ nhất, thứ hai, … Cho nên số lớn thứ *m* thì là phần tử có chỉ số *m* – 1. Nói chung ta có thể hình dung mảng như là dãy các biến (cùng kiểu) liên tiếp, cùng tên, truy xuất bằng vị trí giống như một dãy ghế trong rạp chiếu phim.

Hình 4.1.1 – Hình minh họa mảng (một chiều) a chiều dài n sức chứa MAX



Hàm nhap[13] thì rõ ràng rồi. Hàm này nhập giá trị của người dùng lần lượt vào các phần tử trong mảng theo thứ tự. Lưu ý đến khuôn mẫu dùng vòng lặp `for` khi xử lý mảng. Bằng cách cho biến chạy làm chỉ số của phần tử (hoặc tính ra chỉ số của phần tử) ta có thể lặp lại việc xử lý các phần tử. Chẳng hạn, trong thân của vòng lặp ta xử lý phần tử thứ `i` của mảng `a` tức là `a[i]` thì bằng cách cho `i` chạy từ 0 đến n – 1 (n là chiều dài mảng) thì ta đã xử lý (lần lượt từ đầu đến cuối) tất cả các phần tử của mảng. Trong hàm nhap, Dòng 7 giúp ta nhập số nguyên từ người dùng vào phần tử thứ `i` - 1 của mảng bằng cú pháp tương tự với biến (&<tên biến>) là `&a[i - 1]`. Ở đây như vậy ta đã dùng `a[i - 1]` như là một l-value.

[12] Một số ngôn ngữ như Pascal sẽ đánh chỉ số từ 1. Tình huống này cũng tương tự như cách đánh số con từ thứ 2 trở đi ở Trung Bộ nhưng lại được đánh từ thứ 1 ở Bắc Bộ.
[13] Nhập chứ không phải là náp:)

Sau khi nhập thì mảng sẽ chứa các phần tử theo thứ tự nhập (nghĩa là số nhập đầu tiên là phần tử thứ 0 của mảng, …, số nhập cuối cùng là phần tử thứ n – 1 của mảng). Hàm `sapxep` giúp ta sắp xếp lại vị trí (tức là chỉ số) của các phần tử theo thứ tự giảm dần, nghĩa là phần tử thứ 0 là phần tử có giá trị lớn nhất, …, phần tử thứ n – 1 là phần tử có giá trị nhỏ nhất. Làm sao làm được điều này? (Tức là làm sao sắp xếp một mảng số nguyên?) Có rất nhiều cách làm khác nhau. Ở đây ta chưa vội quan tâm tới điều đó. Bạn chỉ cần chép đúng như vậy cho mã của hàm `sapxep` và hiểu ý nghĩa của hàm đó (tức là nó làm gì? Sắp xếp giảm dần) là được. Nói sơ cách làm này như sau: Xét tất cả các cặp phần tử của mảng, nếu chúng đứng sai chỗ so với thứ tự mong muốn là giảm dần, nghĩa là phần tử có giá trị nhỏ hơn lại đứng trước phần tử có giá trị lớn hơn, thì ta đổi chỗ chúng. Nói rõ hơn là: xét tất cả các cặp vị trí (`i`, `j`) (tức là cặp chỉ số) mà `i` trước `j` (tức là `i < j`), nếu phần tử tại `i` nhỏ hơn phần tử tại `j` (tức là `a[i] < a[j]`) thì ta đổi chỗ hai phần tử (tức tại `i` sẽ chứa `a[j]` còn tại `j` sẽ chứa `a[i]`). Để duyệt qua tất cả các cặp số nguyên (`i`, `j`) mà `i < j` và 0 ≤ `i` < n, 0 ≤ `j` < n (do `i`, `j` là các chỉ số của mảng có chiều dài n) thì ta có dùng khuôn mẫu hai vòng for lồng như trong mã trên.

MỞ RỘNG 4.1 – Trung bình và phương sai của một mẫu dữ liệu số

Một *mẫu dữ liệu* (số) cỡ *n* là một danh sách gồm *n* số thực $x_1, x_2, \ldots, x_n$. Ta gọi giá trị $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1+x_2+\ldots+x_n}{n}$ là *trung bình* của mẫu, giá trị $S^2 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n} = \frac{(x_1-\bar{x})^2+(x_2-\bar{x})^2+\ldots+(x_n-\bar{x})^2}{n}$ là *phương sai* của mẫu và $S = \sqrt{S^2}$ là *độ lệch chuẩn* của mẫu. Các giá trị này cho ta một hình dung đơn giản về mẫu dữ liệu. Giá trị trung bình cho biết giá trị trung tâm (hay trọng tâm) mà các số liệu của mẫu xoay quanh đó còn độ lệch chuẩn (hay phương sai) cho thấy mức độ phân tán (hay xa cách) của các số liệu so với giá trị trung tâm. Độ lệch chuẩn (hay phương sai) càng nhỏ cho thấy các số liệu của mẫu càng bu gần giá trị trung tâm, cũng có nghĩa là chúng bu gần nhau. Ngược lại, độ lệch chuẩn lớn cho thấy một hình ảnh tản mác, xa cách nhau của các số liệu. Đây hiển nhiên là các giá trị quan trọng trong Thống kê. Tuy nhiên ở đây ta đơn giản sẽ là tìm cách tính chúng.

Như đã nói cách xử lý offline thường dễ dàng: ta dùng mảng số thực (kiểu `double`) để chứa (toàn bộ) các số liệu của mẫu (kèm với một biến nguyên *n* cho biết chiều dài mảng và cũng là cỡ mẫu) rồi mới tính trung bình và phương sai của mẫu, chẳng hạn, bằng 2 hàm sau.

Mã 4.1.4 – Các hàm tính trung bình và phương sai mẫu

```
1  double mean(double x[], int n)
2  {
3      double S = 0;
4      for(int i = 0; i < n; i++)
5          S = S + x[i];
6  
7      return S/n;
8  }
9  
10 double var(double x[], int n)
11 {
12     double m = mean(x, n);
13     double S = 0;
14     for(int i = 0; i < n; i++)
15         S = S + (x[i] - m)*(x[i] - m);
16 
17     return S/n;
18 }
```

Hàm `mean` tính trung bình của mẫu số liệu để trong mảng số thực `x` với tham số `n` cho biết cỡ mẫu. Hàm này cài đặt nguyên si công thức tính giá trị trung bình ở trên. Cũng lưu ý là ta thường kí hiệu các số liệu mẫu là x_1, x_2, …, x_n nhưng vì chỉ số mảng trong C bắt đầu từ 0 nên chúng sẽ là `x[0]`, `x[1]`, …, `x[n-1]`. Tương tự, hàm `var`[14] tính phương sai của mẫu theo công thức đã cho. Lưu ý là ta phải tính giá trị trung bình trước và chỉ cần tính một lần như Dòng 12 cho thấy. Bạn hãy ráp vào chương trình, cho nhập mẫu số liệu vào mảng rồi dùng các hàm trên để tính trung bình và phương sai của mẫu số liệu đã nhập. Để tính độ lệch chuẩn, ta tính căn của phương sai.

Như (cũng) đã nói cách xử lý online thường khó khăn hơn, thậm chí là bất khả thi trong một số trường hợp. Ở đây để tính trung bình theo cách online không khó: ta chỉ cần tính tổng $\sum_{i=1}^{n} x_i$ theo cách online và chia cho cỡ mẫu sau cùng để được $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$. Tuy nhiên để tính phương sai thì khác, theo công thức đã cho, ta phải có giá trị trung bình $\bar{x}$ trước khi tính tổng $\sum_{i=1}^{n}(x_i - \bar{x})^2$, mà giá trị này là không thể có khi ta chưa có đủ hết các số liệu của mẫu. Thế ta vẫn ngoan cố muốn tính phương sai theo cách online thì sao? Ta sẽ biến điều không thể thành có thể bằng cách biến đổi tổng trên như sau:

[14] Viết tắt của variance, nghĩa là phương sai.

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n}({x_i}^2 - 2x_i\bar{x} + \bar{x}^2)$$
$$= \sum_{i=1}^{n} {x_i}^2 - 2\bar{x}\sum_{i=1}^{n} x_i + \bar{x}^2 \sum_{i=1}^{n} 1$$
$$= \sum_{i=1}^{n} {x_i}^2 - 2\bar{x}(n\bar{x}) + \bar{x}^2 n = \sum_{i=1}^{n} {x_i}^2 - n\bar{x}^2$$

Do đó $S^2 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n} = \frac{\sum_{i=1}^{n} {x_i}^2}{n} - \bar{x}^2$. Từ đây ta thấy có thể tính tổng $\sum_{i=1}^{n} {x_i}^2$ theo cách tính online và sau cùng chia cho cỡ mẫu và trừ cho bình phương trung bình để được S^2. Hoan hô! Phần còn lại chỉ là cài đặt công thức tính phương sai online như trên. Bạn chạy thử chương trình tính trung bình và phương sai theo cách online sau. Nhớ nghiền ngẫm để thấy hết cái hay nhé. Nó sẽ giúp bạn tăng công lực đáng kể đó.

Mã 4.1.5 – Chương trình tính trung bình và phương sai mẫu theo cách online

```
int main()
{
    double x, S = 0, S2 = 0;
    int n = 0;
    while(1)
    {
        int cmd;
        printf("\nBan muon (1 - Nhap so lieu, 2 - Tra gia tri, 3 - Dung): ");
        scanf("%d", &cmd);

        if(cmd == 1)
        {
            printf("Nhap so lieu tiep theo: ");
            scanf("%lf", &x);
            S += x;
            S2 += x*x;
            n++;
        }
        else if(cmd == 2)
        {
            printf("Co mau: %d\n", n);
            printf("Trung binh mau: %lf\n", S/n);
            printf("Phuong sai mau: %lf\n", S2/n - (S/n)*(S/n));
        }
        else
        {
            printf("Bye!");
```

```
                break;
            }
        }

        return 0;
}
```

BÀI TẬP

Bt 4.1.1 Hãy xem lại phần Mở rộng 1.4. Một đa thức *P* bậc *n* theo biến *x* là biểu thức *P*(*x*) có dạng $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ trong đó a_0, a_1, …, a_n là các số thực cho trước, gọi là các hệ số của đa thức, với $a_n \neq 0$. Bằng cách dùng mảng để chứa các hệ số của đa thức, hãy viết chương trình cho nhập một đa thức *P*, giá trị thực *x* và tính *P*(*x*) bằng phương pháp Horner.

Bt 4.1.2 Hãy xem lại Bài tập 2.5.4: tìm tất cả các số nguyên tố không quá số nguyên dương *n* được cho. Có một thuật toán (cách làm) rất hiệu quả là "Sàng Eratosthenes", bạn hãy tham khảo thuật toán này trên trang Wikipedia[15] và dùng mảng để cài đặt nó. Từ đó hãy làm lại Bài tập 2.5.5 để phân tích một số nguyên lớn hơn 1 ra tích các thừa số nguyên tố.

Bt 4.1.3 Hãy xem lại phần Mở rộng 2.6. Bạn hãy viết lại thật ngắn gọn chương trình chơi Bầu Cua (Mã 2.6.4) bằng cách dùng các mảng phù hợp cho các đại lượng liên quan đến tôm, cua, …, nai. (Bạn sẽ ngạc nhiên về sự cô đọng của mã mới so với mã cũ.)

Bt 4.1.4 Hãy xem lại Bài tập 3.2.3(f), nếu được dùng mảng thì việc tính số hạng Fibonacci thứ *n* (f_n) có dễ hơn không?

Bt 4.1.5 Hãy xem lại phần Mở rộng 3.4. Hàm có số lượng đối số tùy ý là một cách để viết các hàm có thể xử lý trên nhiều dữ liệu với số lượng tùy ý. Cách khác để viết các hàm như vậy là dùng tham số mảng cho hàm vì mảng có thể chứa nhiều dữ liệu (cùng kiểu) với số lượng tùy ý (trong sức chứa). Hãy viết lại hàm tính tổng *n* số nguyên (Mã 3.4.12) bằng cách dùng tham số mảng. Tương tự cho các Bài tập 3.4.4, Bài tập 3.4.5, Bài tập 3.4.6 và Bài tập 3.4.7.

Bt 4.1.6 Hãy xem lại phần Mở rộng 4.1. Cho một mẫu dữ liệu, ta gọi *trung vị* (median) của mẫu là giá trị "ở giữa" hay giá trị "chia đôi dữ liệu" của mẫu. Tương tự giá trị trung bình, trung vị là một con số phản ánh trung tâm của số liệu. Hãy tham khảo tài liệu để hiểu rõ khái niệm trung vị và viết hàm tính giá trị này của một mẫu dữ liệu (prototype tương tự hàm `mean` hay `var` của Mã 4.1.4).

[15] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Bt 4.1.7 Kho tàng bài tập về mảng của C rất lớn với các dạng chủ yếu là: nhập, xuất, tạo, tính, kiểm tra, đếm, tìm, thêm, xóa, sửa và sắp xếp. Bạn hãy tham khảo một vài nguồn bài tập như vậy và làm vài bài trong đó[16].

Bt 4.1.8 Trong bài học trên, để sắp xếp các phần tử trong mảng (hàm sapxep trong Mã 4.1.3) ta đã dùng cách *sắp xếp đổi chỗ* (Interchange Sort). Hãy tham khảo và cài đặt các thuật toán sắp xếp đơn giản khác là *sắp xếp chọn* (Selection Sort[17]), sắp xếp chèn (Insertion Sort[18]) và *sắp xếp nổi bọt* (Bubble Sort[19]). Thật sự thì có 101 cách (thuật toán) sắp xếp khác nhau[20]. Điều này cũng cho thấy bài toán sắp xếp là rất quan trọng.

Mảng là công cụ vô giá của C!

---

[16] Các bài tập như vậy có đầy trên mạng, bạn làm thử vài bài khó thôi nhé, không cần tốn nhiều thời gian.

[17] https://en.wikipedia.org/wiki/Selection_sort

[18] https://en.wikipedia.org/wiki/Insertion_sort

[19] https://en.wikipedia.org/wiki/Bubble_sort

[20] Tham khảo https://en.wikipedia.org/wiki/Sorting_algorithm. (Mà sao chẳng thấy Interchange Sort nhỉ?)

BÀI 4.2

Trong bài này ta sẽ học vẽ bằng các kí tự. Trong phần đầu của Bài 3.2 ta đã vẽ một hình vuông hay một hình tam giác vuông cân ra màn hình. Bạn hãy viết và chạy lại thử nhé. Hơn nữa để việc xuất ra được đẹp ta có thể chỉnh kích thước các kí tự trong cửa sổ Console cho khoảng cách ngang bằng khoảng cách dọc hay tăng kích thước của hình để khi tăng *n* ta vẫn không bị "vỡ hình" như trong Phụ lục A.4.

Trường hợp hình phức tạp hơn thì ta rất khó dùng khung chương trình trên do không dễ tính được cách xuất các kí tự trên mỗi dòng. Chẳng hạn để xuất ra hình tròn, bạn có thấy được qui tắc xuất ra các kí tự trên dòng thứ *i* không? Lí do của khó khăn này là ta đã xuất tuần tự từng dòng: xử lý và xuất trên dòng này xong rồi qua xử lý và xuất trên dòng tiếp theo. Đó là cách *xuất online*: vừa xử lý vừa xuất. Tưởng tượng là hình của ta có 100 dòng, và cần 1 giây để xử lý từng dòng thì sau 1 giây đầu ta có kết quả của dòng thứ nhất, sau 1 giây nữa ta có thêm kết quả của dòng thứ hai, … sau 50 giây đầu ta có được nửa bức hình và sau 100 giây ta có đầy đủ bức hình. Cách làm này (xuất online) như vậy có lợi là ta có hình ảnh từ từ trong quá trình xuất[1] nhưng bù lại sẽ rất khó khăn cho việc xử lý. Một cách làm khác là ta có thể tính hết tất cả các kí tự của hình xong rồi ta mới xuất một thể. Cách làm này gọi là *xuất offline*: xử lý xong hết rồi mới xuất. Cách này có nhược điểm là không có kết quả xuất tức thời (do phải chờ xử lý, mà thời gian xử lý thường lâu hơn thời gian xuất). Tưởng tượng là hình của ta có 100 dòng, và cần 99 giây để xử lý toàn bộ và 1 giây để xuất toàn bộ (thời gian xử lý một cách toàn bộ thường ít hơn tổng thời gian xử lý riêng lẻ và thời gian xuất (chỉ xuất) thì nhanh hơn thời gian xử lý) thì sau 1 giây đầu ta không có gì cả, sau 1 giây nữa ta cũng không có gì cả, …, sau 50 giây đầu ta cũng không có gì cả, … sau 99 giây cũng chưa có gì cả, thêm 1 giây nữa, đùng một phát ta có toàn bộ bức hình. Bù lại, cách xử lý offline này sẽ dễ dàng hơn nhiều do ta không cần phải tính cách xử lý riêng lẻ cho từng dòng, hết dòng này đến dòng khác mà xử lý toàn bộ.

Để có thể xuất offline thì ta cần một vùng nhớ chứa các kí tự cần kết xuất của bức hình, ta xử lý (điền kí tự) toàn bộ trên vùng nhớ này xong rồi thì mới xuất ra. Vùng nhớ này thường được gọi là *bộ đệm*. Cách xử lý offline như vậy còn có nhược điểm là tốn thêm vùng nhớ. Trong trường hợp này do hình của ta là hình hai chiều (ngang và dọc) nên vùng nhớ này là *vùng nhớ*

[1] Liên hệ cách hiển thị của hình ảnh trên web để dễ hình dung nhé.

hai chiều. Ta đã gặp vùng nhớ 1 chiều trong bài trước, đó chính là mảng (một chiều). Vậy ta cần *mảng hai chiều* trong trường hợp này. Bài này là về mảng hai chiều vì hình ta vẽ là hai chiều. Bạn cũng cần nắm kĩ Bài 4.1, mảng một chiều, vì mảng hai chiều là mở rộng tự nhiên của mảng một chiều (thêm một chiều nữa thôi mà:)).

Khung chương trình của ta sẽ như sau.

Mã 4.2.1 – Khung chương trình vẽ hình offline

```
1  int main()
2  {
3      char buff[MAX][MAX];
4      int n = 20;
5      fillblank(' ', buff, n);
6  
7      // Điền các kí tự của hình vào bộ đệm buff
8  
9      display(buff, n);
10 
11     return 0;
12 }
```

Nếu như mảng một chiều cho phép ta lưu trữ nhiều dữ liệu của một kiểu nào đó được bố trí theo một chiều (có thể tưởng tượng như một dòng) thì mảng hai chiều cho phép ta lưu trữ nhiều dữ liệu của một kiểu nào đó được bố trí theo hai chiều (có thể tưởng tượng như một bảng chữ nhật). Trong đó chiều dọc, từ trên xuống dưới theo dòng, là chiều thứ nhất và chiều ngang, từ trái qua phải theo cột, là chiều thứ hai. Ta còn gọi mảng hai chiều là bảng hay ma trận. Khai báo mảng 2 chiều là mở rộng tự nhiên của khai báo mảng 1 chiều:

<kiểu của phần tử> <tên mảng 2 chiều>[<số dòng tối đa>][<số cột tối đa>];

Ở đây ta cần thao tác trên một bảng (tức là mảng hai chiều) các kí tự theo dòng, cột nên ta cần bảng các kí tự. Một cách tổng quát thì đó là *bảng chữ nhật* (số dòng và số cột có thể khác nhau) nhưng để đơn giản, hình của ta được bố trí trong một khung vuông nên ta sẽ dùng một *bảng vuông* các kí tự, tức là mảng hai chiều các kí tự với số dòng bằng số cột. Mảng của ta tên là `buff`[2] được khai báo như Dòng 3, trong đó `MAX` là hằng tượng trưng cho biết kích thước của vùng đệm là kích thước tối đa mỗi chiều của hình, tức là số dòng và số cột tối đa. Ta cần thêm chỉ thị `#define` vào đầu chương trình (sau các chỉ thị `#include`), chẳng hạn như:

```
#define MAX 100
```

[2] Viết tắt của buffer – vùng nhớ đệm.

Cũng như trong mảng một chiều, các con số trong khai báo mảng cho biết *sức chứa* của mảng, ở đây là số tối đa các dòng, cột mà bảng đó có thể chứa, còn số dòng, cột thực sự thì ta phải dùng thêm biến để lưu trữ chúng. Trường hợp tổng quát ta có bảng chữ nhật *m* dòng *n* cột, khi đó ta dùng biến nguyên `m` cho số dòng và `n` cho số cột[3] và $m \times n$ còn được gọi là *kích thước* của bảng. Ở trên ta dùng biến `n` chứa số cột (và cũng là số dòng) thực sự của bảng (cũng là của hình), tức là bảng có kích thước `n` × `n`. Dòng 4 đặt giá trị cho `n` là 20 nhưng hiển nhiên là ta cũng có thể cho người dùng nhập giá trị cho `n`.

Hàm `fillblank` điền các kí tự trắng ("màu nền") vào `buff` (Dòng 5). Sau đó ta điền các kí tự của hình vào `buff` (Dòng 7) và hàm `display` sẽ xuất các kí tự trong `buff` ra màn hình (Dòng 9). Hiển nhiên là ta cần hiện thực tất cả điều này.

Trước hết, hàm `fillblank` có thể được viết như sau.

Mã 4.2.2 – Hàm `fillblank`

```
1 void fillblank(char c, char a[][MAX], int n)
2 {
3     for(int i = 0; i < n; i++)
4         for(int j = 0; j < n; j++)
5             a[i][j] = c;
6 }
```

Hàm này điền kí tự trắng `c` vào toàn bộ bảng kí tự `a`. Bạn hãy xem prototype để biết cách khai báo tham số mảng hai chiều. Lưu ý là ta cần cho biết số cột tối đa (ở đây là `MAX`) trong khai báo tham số `a` mà không cần cho biết số dòng tối đa[4]. Cũng như mảng một chiều ta cần tham số chứa kích thước mảng đi kèm, ở đây là `n`, cho kích thước `n` × `n`.

Ta đã dùng chỉ số khi truy cập phần tử của mảng một chiều. Với mảng hai chiều ta cần hai chỉ số: *chỉ số dòng* (dòng thứ mấy) và *chỉ số cột* (cột thứ mấy). Cũng vậy các chỉ số đều được tính từ 0. Hơn nữa chỉ số dòng viết trước, chỉ số cột viết sau. Như vậy cú pháp truy cập vào phần tử của mảng 2 chiều là:

<tên mảng>[<chỉ số dòng>][<chỉ số cột>]

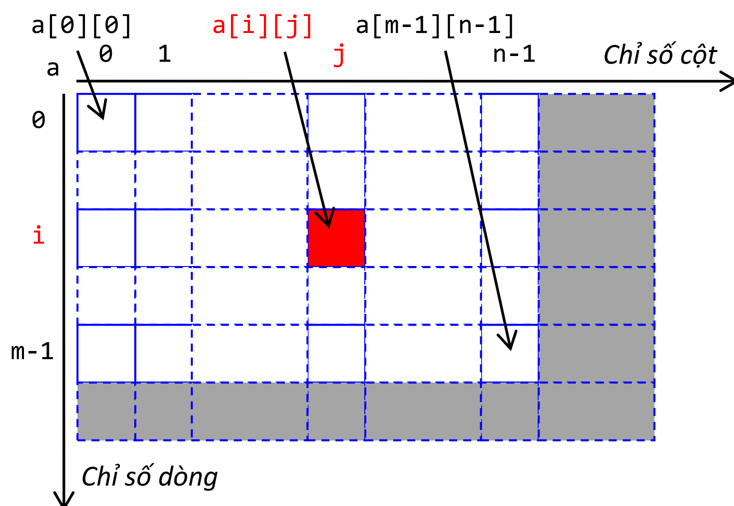
Biểu thức truy cập trên là một l-value (tương đương với một biến). Ở trên để truy cập vào phần tử (kí tự) ở dòng thứ `i` cột thứ `j` của bảng `a` ta viết `a[i][j]` (Dòng 5). Để duyệt qua tất cả các phần tử của bảng kích thước $m \times n$, ta cần duyệt qua tất cả các cặp *chỉ số* (i, j) với $0 \le i \le m - 1$, $0 \le j \le n -$

[3] Dĩ nhiên bạn có thể dùng các tên biến khác nhưng đó là thói quen của nhiều lập trình viên.

[4] Ta sẽ rõ lý do sau, nhớ là trong khai báo tham số mảng một chiều thì ta không cần cho biết sức chứa.

1. Ta có thể dùng khuôn mẫu 2 vòng `for` lồng nhau như trong đoạn mã trên cho bảng a kích thước n × n. Cũng theo thói quen mà ta hay dùng biến lặp `i` cho chỉ số dòng và biến lặp `j` cho chỉ số cột. Bạn nhớ lại khuôn mẫu dùng vòng `for` để duyệt qua các phần tử của mảng một chiều.

Hình 4.2.1 – Hình minh họa mảng hai chiều a kích thước m×n[5]



Hàm `display` có thể được viết như sau.

Mã 4.2.3 – Hàm `display`

```
1 void display(char a[][MAX], int n)
2 {
3     for(int i = 0; i < n; i++)
4     {
5         for(int j = 0; j < n; j++)
6             printf("%c", a[i][j]);
7         printf("\n");
8     }
9 }
```

Không có thêm gì mới ở hàm này. Ta duyệt qua và xuất các kí tự trên từng dòng, xuất xong dòng trên thì xuống dòng (Dòng 7) rồi xuất dòng dưới, điều này là không thể tránh được do bản chất của việc xuất (hàm `printf`) là như vậy. Tuy nhiên ở đây là xuất không chứ không có xử lý (không tính các kí tự cần xuất) nên rất nhanh[6].

[5] Bạn đối chiếu với hình minh họa của mảng một chiều ở Bài 4.1 (Hình 4.1.1).

[6] Ta có thể xuất nhanh hơn nữa bằng cách xuất nguyên từng dòng chứ không phải từng kí tự trong dòng, ta sẽ tìm hiểu thêm trong bài về chuỗi.

Công việc của ta chủ yếu là ở việc xử lý, điền các kí tự của hình vào bảng `buff`. Chẳng hạn hàm sau sẽ điền các kí tự của hình tròn.

Mã 4.2.4 – Hàm `circle` "vẽ" hình tròn vào bộ đệm

```
void circle(char c, char a[][MAX],
                        int x0, int y0, int r)
{
    for(int i = -r; i <= r; i++)
        for(int j = -r; j <= r; j++)
            if(i * i + j * j <= r * r)
                a[x0 + i][y0 + j] = c;
}
```

Hàm trên điền kí tự `c` vào bảng `a` theo hình tròn tâm (`x0`, `y0`), bán kính `r`. Như ta thấy việc tính từng dòng cho hình tròn rất khó nhưng việc xử lý toàn bộ thì dễ hơn. Cho hình tròn tâm $O(0, 0)$ bán kính r. Điểm $A(x, y)$ nằm trong hình tròn khi và chỉ khi $OA \leq r$, tức là $OA^2 \leq r^2$, tức là $(x^2 + y^2) \leq r^2$. Sau đó ta cần tịnh tiến mọi điểm của hình tròn có tâm tại $O(0, 0)$ đến tâm $C(x_0, y_0)$, nghĩa là tịnh tiến mọi điểm của hình tròn theo vector (x_0, y_0). Như vậy điểm $A(x, y)$ trong hình tròn sẽ thành điểm $A'(x_0 + x, y_0 + y)$. Đây chính là nội dung của Mã 4.2.4.

Để vẽ hình tròn bạn thêm hàm `circle` trên vào mã và thay Dòng 7 của khung chương trình (Mã 4.2.1) bằng lời gọi hàm tương ứng, chẳng hạn:

```
circle('*', buff, n/2, n/2, n/2);
```

sẽ vẽ hình tròn có tâm ngay chính giữa khung và bán kính là nửa khung bằng kí tự sao. Để kết xuất đẹp hãy cho `n` giá trị lẻ (khi đó tâm sẽ nằm ngay chính giữa). Ấn tượng hơn ta có thể tô nền bằng kí tự sao (gọi `fillblank` với đối số đầu tiên là `'*'`) và vẽ hình tròn với kí tự trắng (gọi `circle` với đối số đầu tiên là `' '`). Bạn cũng có thể gọi nhiều lần `circle` để vẽ nhiều hình tròn với tâm, bán kính và kí tự điền khác nhau.

Để minh họa sự dễ dàng với cách xử lý này (xử lý offline) ta sẽ vẽ hình thoi đều (là hình vuông xoay 90^0) bằng hàm sau.

Mã 4.2.5 – Hàm `rhombus` "vẽ" hình thoi đều vào bộ đệm

```
void rhombus(char c, char a[][MAX],
                         int x0, int y0, int r)
{
    int i, j;
    for(i = -r; i <= r; i++)
        for(j = -r; j <= r; j++)
            if(abs(i) + abs(j) <= r)
                a[x0 + i][y0 + j] = c;
}
```

Hàm này vẽ hình thoi đều tâm (x0, y0), bán kính r với kí tự c. Tương tự hàm circle, bạn hãy nghiên cứu nội dung của hàm rhombus. Hàm abs trả về trị tuyệt đối của một số nguyên và có thể được viết như sau:

```
int abs(int d)
{
    return d >= 0 ? d : -d;
}
```

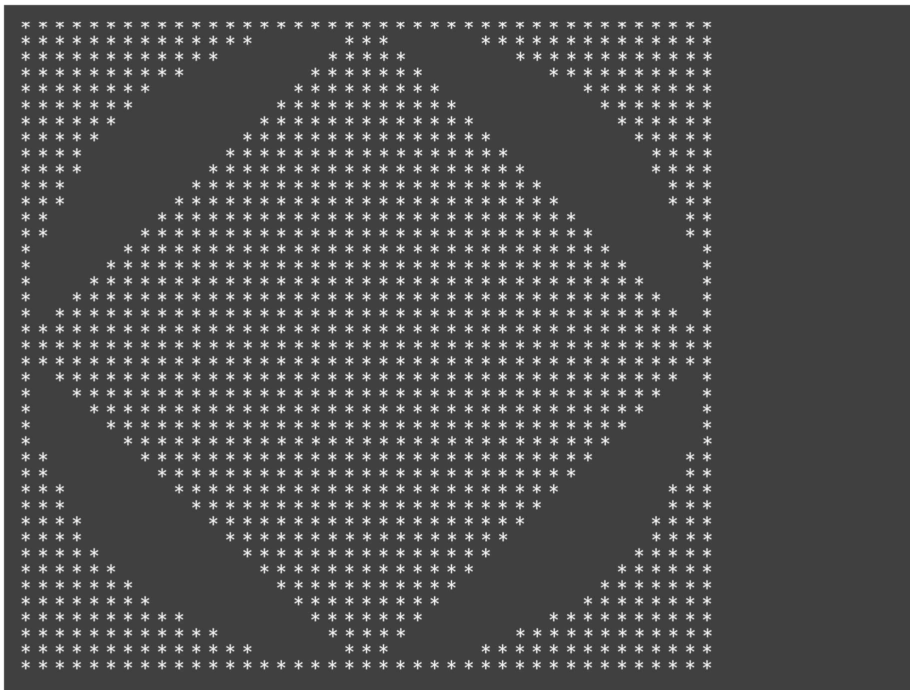
Để vẽ hình thoi ngay chính giữa khung và bán kính là nửa khung bằng kí tự sao ta có thể điền buff bằng cách gọi:

```
rhombus('*', buff, n/2, n/2, n/2);
```

Dĩ nhiên ta có thể gọi nhiều lần để kết hợp vẽ các hình. Chẳng hạn trên nền kí tự sao ta đục hình tròn (kí tự trắng) rồi tô hình thoi (kí tự sao) như sau:

```
int n = 41;
fillblank('*', buff, n);
circle(' ', buff, n/2, n/2, n/2);
rhombus('*', buff, n/2, n/2, n/2);
display(buff, n);
```

Bạn chạy để xem kết quả (xem Phụ lục A.4 để biết cách mở rộng kích thước cửa sổ Console khi n lớn).



Lưu ý là hàm `rhombus` được gọi sau hàm `circle` vì hình tròn to hơn hình thoi. Nếu gọi hàm `circle` sau hàm `rhombus` thì thực ra ta chỉ vẽ hình tròn.

MỞ RỘNG 4.2 – Hình fractal thảm Sierpinski

Để thuyết phục bạn rằng cách xử lý offline thường dễ hơn online thì ta sẽ dùng khung chương trình trên để vẽ một hình khá phức tạp gọi là *thảm Sierpinski* (Sierpinski carpet). Đây là một hình fractal mà qui luật xuất trên từng dòng là khó thấy. Bạn hãy tham khảo trên trang Wikipedia để biết về Fractal[7] và Sierpinski carpet[8]. Các hình fractal có bản chất *đệ qui* nên ta cài đặt hàm vẽ thảm Sierpinski bằng đệ qui như sau[9].

Mã 4.2.6 – Hàm "vẽ" thảm Sierpinski

```
void Sierpinski(char c, char a[][MAX],
                        int x0, int y0, int w)
{
    if(w == 0)
        return;

    for(int i = 0; i < w/3; i++)
        for(int j = 0; j < w/3; j++)
            a[x0 + w/3 + i][y0 + w/3 + j] = c;

    Sierpinski(c, a, x0, y0, w/3);
    Sierpinski(c, a, x0, y0  + w/3, w/3);
    Sierpinski(c, a, x0, y0 + 2*w/3, w/3);
    Sierpinski(c, a, x0 + w/3, y0, w/3);
    Sierpinski(c, a, x0 + w/3, y0 + 2*w/3, w/3);
    Sierpinski(c, a, x0 + 2*w/3, y0, w/3);
    Sierpinski(c, a, x0 + 2*w/3, y0  + w/3, w/3);
    Sierpinski(c, a, x0 + 2*w/3, y0 + 2*w/3, w/3);
}
```

Hàm này vẽ thảm Sierpinski bắt đầu tại góc trên trái (`x0`, `y0`) với kích thước `w` bằng kí tự trắng `c`. Hàm `Sierpinski` vẽ thảm Sierpinski theo bản chất đệ qui của nó: từ hình vuông kích thước `w`, chia ra 9 hình vuông con kích thước `w/3`, đục (bỏ) đi hình vuông ở giữa (tức là điền đầy kí tự trắng `c`) và vẽ 8 thảm Sierpinski thu nhỏ (tỉ lệ 1/3) vào 8 hình vuông còn lại xung quanh.

Để hình vẽ đẹp, hãy đặt kích thước là lũy thừa của 3 và vẽ bằng kí tự

[7] https://en.wikipedia.org/wiki/Fractal
[8] https://en.wikipedia.org/wiki/Sierpinski_carpet
[9] Ta sẽ tìm hiểu về đệ qui sau.

trắng, chẳng hạn gọi:

```
int n = 81;
fillblank('*', buff, n);
Sierpinski(' ', buff, 0, 0, n);
display(buff, n);
```

Bạn có thấy vẻ đẹp của hình này không? (Nó sẽ ấn tượng hơn nhiều nếu độ phân giải của chúng ta cao hơn, nghĩa là n lớn và từng kí tự nhỏ lại). Ở đây tôi muốn bạn ấn tượng với sự dễ dàng của cách làm offline so với online và ấn tượng về cái gọi là đệ qui (mà ta sẽ tìm hiểu sau). Về các hình fractal ta có thể gặp lại đâu đó.

BÀI TẬP

Bt 4.2.1 Hãy xem lại Bài 2.7. Như đã nói thì mã của Bài 2.7 quá dông dài và kém tự nhiên do hạn chế của việc xuất online. Bằng cách dùng mảng hai chiều với kĩ thuật xuất offline như trong bài học, bạn hãy viết lại chương trình của Bài 2.7 thật ngắn gọn.

Bt 4.2.2 Làm lại Bài tập 2.7.2 và Bài tập 2.7.3 bằng cách dùng mảng hai chiều với kĩ thuật xuất offline.

Bt 4.2.3 Hãy xem lại Bài tập 3.2.2. Bạn đã rất vất vả để xuất ra các hình trong bài tập đó do hạn chế của việc xuất online. Bằng cách dùng mảng hai chiều với kĩ thuật xuất offline, bạn hãy làm lại Bài tập này. (Bạn sẽ ngạc nhiên về sự cô đọng, dễ dàng và tự nhiên của mã mới so với mã cũ.) (*Gợi ý*: để ý bạn sẽ thấy rằng mặc dù các hình khá phức tạp nhưng nó được tạo nên từ chỉ 4 dạng hình cơ bản, 4 nửa tam giác vuông, đặt tại các vị trí khác nhau.)

Bt 4.2.4 Hãy xem lại Bài tập 3.2.3(g), nếu được dùng mảng hai chiều thì việc tính hàm Ackermann ($A(m, n)$) có dễ hơn không?

Bt 4.2.5 Hãy xem lại Bài tập 3.2.4. Bạn đã rất vất vả để xuất ra tam giác Pascal (đặc biệt là khi canh giữa). Bạn hãy làm lại Bài tập này.

Bt 4.2.6 Làm lại Bài tập 3.3.3 bằng cách dùng mảng hai chiều.

Bt 4.2.7 Làm tương tự Bài tập 4.1.7 cho mảng hai chiều. Hơn nữa vì các dòng, các cột hay đường chéo của mảng hai chiều “có thể xem như mảng một chiều” nên có thể yêu cầu xử lý riêng cho chúng. Chẳng hạn: tính tổng từng dòng hay từng cột của mảng hai chiều.

Bt 4.2.8 Một phần tử của mảng hai chiều các số được gọi là “điểm yên ngựa” nếu phần tử đó là nhỏ nhất trong dòng và lớn nhất trong cột. Tìm tất cả các điểm yên ngựa của một mảng.

Mảng là công cụ vô giá của C!

BÀI 4.3

Hãy xem cách sắp xếp một mảng các số nguyên (nghĩa là một danh sách các số) trong hàm `sapxep` ở Bài 4.1 (Mã 4.1.3). Sắp giảm dần, nghĩa là các số lớn đứng trước các số nhỏ. Như bạn đã biết, có 101 cách sắp xếp như vậy[1]. Cách mà tôi đã dùng có thể xem là cách đơn giản nhất. Cách sắp xếp này được gọi là *sắp xếp đổi chỗ* vì nó dựa vào việc duyệt qua các cặp vị trí và đổi chỗ nếu cặp này không đúng theo thứ tự sắp xếp. Một cặp vị trí như vậy được gọi là *cặp nghịch thế*. Chẳng hạn khi sắp giảm dần thì cặp vị trí (i, j) với i trước j (nghĩa là $i < j$) mà giá trị tại i lại nhỏ hơn giá trị tại j (nghĩa là $a[i] < a[j]$) thì cặp này là nghịch thế. Để rõ hơn, hàm sắp xếp trên có thể được viết lại như sau.

Mã 4.3.1 – Hàm sắp xếp viết rõ hơn

```
void sapxep(int a[], int n) {
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(nghichthe(a, i, j)) {
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
```

Ở trên, ta đã duyệt qua tất cả các cặp vị trí có thể (`i`, `j`) với (0 ≤ `i` < n, 0 ≤ `j` < n) và kiểm tra nếu (`i`, `j`) là cặp nghịch thế thì ta hoán đổi giá trị ở vị trí `i` với vị trí `j`. Hàm `nghichthe` cho biết vị trí `i`, `j` trên mảng `a` có là cặp nghịch thế không. Tùy theo cách ta muốn sắp xếp mà ta sẽ định nghĩa hàm này. Chẳng hạn nếu muốn sắp giảm thì cặp vị trí là nghịch thế được xác định như trên và có thể được cài đặt như sau.

Mã 4.3.2 – Hàm kiểm tra nghịch thế cho sắp xếp giảm dần

```
int nghichthe(int a[], int p1, int p2) {
    if((p1 < p2) && (a[p1] < a[p2]))
        return 1;

    return 0;
}
```

[1] Ta sẽ bàn sâu hơn trong một bí kíp khác.

Hàm này trả về giá trị đúng (1) khi cặp vị trí (p1, p2) là cặp nghịch thế trên mảng a và trả về giá trị sai (0) khi không là cặp nghịch thế.

Cách cài đặt mới của ta là rõ ràng và tổng quát hơn (bù lại sẽ chạy chậm hơn một chút). Thật vậy bằng cách định nghĩa hàm `nghichthe` khác nhau ta có thể dễ dàng sắp xếp theo các cách khác nhau. Chẳng hạn để sắp xếp tăng dần ta có thể viết hàm nghịch thế như sau (mà ý của nó là cặp (p1, p2) là nghịch thế nếu p1 trước p2 mà giá trị tại p1 lại lớn hơn p2).

Mã 4.3.3 – Hàm kiểm tra nghịch thế cho sắp xếp tăng dần

```
int nghichthe(int a[], int p1, int p2) {
    if((p1 < p2) && (a[p1] > a[p2]))
        return 1;

    return 0;
}
```

Bạn có thể viết hàm nghịch thế để sắp xếp các số nguyên sao cho: số chẵn đứng trước số lẻ, số chẵn giảm dần, số lẻ tăng dần không? Bạn nghiên cứu mã sau nhé (mã viết hơi cô đọng nên hơi khó hình dung).

Mã 4.3.4 – Hàm kiểm tra nghịch thế cho sắp xếp "chẵn trước lẻ, chẵn giảm dần, lẻ tăng dần"

```
int nghichthe(int a[], int p1, int p2) {
    if(p1 < p2)
        if(a[p1] % 2 != 0) {
            if(a[p2] % 2 == 0)
                return 1;
            else if(a[p1] > a[p2])
                return 1;
        }
        else if(a[p2] % 2 == 0 && a[p1] < a[p2])
            return 1;

    return 0;
}
```

Công việc của ta trong bài này là sắp xếp trên mảng hai chiều các số nguyên. Ta có thể dùng lại khuôn mẫu trên với khác biệt là vị trí là cặp chỉ số (dòng, cột). Hàm `main` được viết như sau.

Mã 4.3.5 – Khung chương trình sắp xếp trên mảng hai chiều

```
int main() {
    int m, n, a[100][100];
    printf("Nhap kich thuoc mang: ");
    scanf("%d %d", &m, &n);

    nhap(a, m, n);
```

```
    printf("Mang truoc khi sap:\n");
    xuat(a, m, n);

    sapxep(a, m, n);
    printf("Mang sau khi sap:\n");
    xuat(a, m, n);

    return 0;
}
```

Hàm nhập và xuất mảng có thể được viết như sau.

Mã 4.3.6 – Hàm nhập và xuất mảng hai chiều

```
void nhap(int a[][100], int m, int n) {
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++) {
            printf("Nhap a[%d][%d]: ", i, j);
            scanf("%d", &a[i][j]);
        }
}

void xuat(int a[][100], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
}
```

Tương tự trường hợp mảng một chiều ta có hàm sắp xếp như sau.

Mã 4.3.7 – Hàm sắp trên mảng hai chiều

```
void sapxep(int a[][100], int m, int n) {
    for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)
    for(int k = 0; k < m; k++)
    for(int l = 0; l < n; l++)
        if(nghichthe(a, i, j, k, l)) {
            int t = a[i][j];
            a[i][j] = a[k][l];
            a[k][l] = t;
        }
}
```

Mỗi vị trí là cặp chỉ số gồm (dòng, cột) do đó để xét tất cả các cặp vị trí ta dùng 4 vòng `for`. Cũng vậy, tùy cách ta sắp xếp mà ta định nghĩa hàm nghịch thế khác nhau. Ở đây ta muốn sắp “tăng theo dòng” nghĩa là tăng dần, hết dòng 1 đến dòng 2, … nên hàm nghịch thế được viết như sau.

Mã 4.3.8 – Hàm kiểm tra nghịch thế cho "sắp tăng theo dòng"

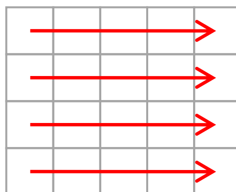
```
int nghichthe(int a[][100], int d1, int c1,
                            int d2, int c2) {
    if((d1 < d2 || (d1 == d2 && c1 < c2))
            && (a[d1][c1] > a[d2][c2]))
        return 1;

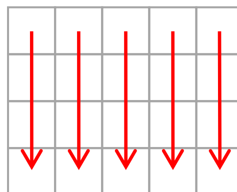
    return 0;
}
```

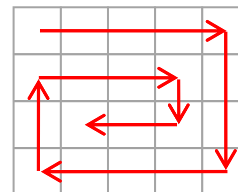
Nếu vị trí (d1, c1) "đứng trước" vị trí (d2, c2) mà giá trị tại (d1, c1) lớn hơn giá trị tại (d1, c1) thì là nghịch thế. Ta có (d1, c1) đứng trước (d2, c2) nếu d1 trước d2 (d1 < d2) hay cùng một dòng (d1 = d2) mà c1 trước c2 (c1 < c2).

Bạn có thể viết hàm nghịch thế để sắp xếp "tăng theo cột" nghĩa là tăng dần hết cột 1 đến cột 2, … hay "tăng theo đường chéo" hay không? Đây cũng không phải là câu đố của bài học này. Câu đố là bạn có thể sắp "tăng theo đường xoắn ốc" như hình sau hay không?

Hình 4.3.1 – Hình minh họa các yêu cầu sắp xếp trên mảng hai chiều


tăng theo dòng


tăng theo cột


tăng theo đường xoắn ốc

Việc sắp xếp trực tiếp trên mảng như ta đã làm gọi là *xử lý in-place* (*xử lý tại chỗ*) và nó có thể gây khó khăn trong vài trường hợp. Ta có cách tiếp cận dễ dàng hơn đó là *xử lý out-of-place*: ta sẽ chép dữ liệu vào chỗ dễ xử lý hơn, sau khi xử lý, ta lại chép dữ liệu về chỗ cũ. Rõ ràng cách này sẽ tốn bộ nhớ hơn (tốn thêm chỗ tạm chứa dữ liệu) và dài dòng hơn nhưng thường dễ dàng hơn. Ở đây do việc sắp xếp trên mảng hai chiều là khó, ta sẽ chép toàn bộ các phần tử của mảng hai chiều sang một mảng một chiều tạm, sau khi sắp xếp trên mảng một chiều (việc này dễ hơn so với sắp xếp trên mảng hai chiều) ta lại chép dữ liệu về mảng hai chiều ban đầu. Hàm `main` của khung chương trình mới như sau.

Mã 4.3.9 – Khung chương trình sắp xếp out-of-place

```
int main() {
    int m, n, a[100][100];
    printf("Nhap kich thuoc mang: ");
    scanf("%d %d", &m, &n);

    nhap(a, m, n);
```

```
    printf("Mang truoc khi sap:\n");
    xuat(a, m, n);

    int x[100*100];
    copyto(a, m, n, x);
    sapxep(x, m * n);
    copyfrom(a, m, n, x);
    printf("Mang sau khi sap:\n");
    xuat(a, m, n);

    return 0;
}
```

Hàm chép dữ liệu sang mảng một chiều có thể làm đơn giản như sau.

Mã 4.3.10 – Hàm chép dữ liệu từ mảng hai chiều sang mảng một chiều

```
void copyto(int a[][100], int m, int n, int x[]) {
    int p = 0;
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            x[p++] = a[i][j];
}
```

Hàm sắp xếp trên mảng một chiều (ở đây là sắp tăng) có thể viết đơn giản như trước đây. Nhớ là mảng một chiều của chúng ta bây giờ chứa m×n phần tử (chép từ mảng 2 chiều kích thước m×n).

Hàm chép dữ liệu từ mảng một chiều sang mảng hai chiều có thể làm như sau.

Mã 4.3.11 – Hàm chép dữ liệu từ mảng một chiều sang mảng hai chiều

```
void copyfrom(int a[][100], int m, int n, int x[]) {
    int p = 0, i = 0, j = 0;
    int l = 0, r = n - 1, t = 0, b = m - 1;
    int dir = RIGHT;
    do {
        a[i][j] = x[p];
        p++;
    }
    while(move(i, j, dir, l, r, t, b));
}
```

Chiến lược của ta là: chạy và chép tương ứng từ mảng một chiều xuống mảng hai chiều. p là vị trí chạy trên mảng một chiều x, (i, j) là vị trí (tương ứng dòng, cột) chạy trên mảng hai chiều a. Ta khởi động vị trí bắt đầu chạy với p = 0 (đầu mảng một chiều) và i = 0, j = 0 (góc trên trái của mảng hai chiều). Chạy trên mảng một chiều rất đơn giản: p chạy từ trước ra sau, nghĩa là p tăng dần (p++). Do ta muốn sắp theo đường xoắn ốc nên trên mảng hai chiều ta sẽ chạy theo đường xoắn ốc, do đó việc chạy khá phức tạp. Biến

dir cho biết hướng đang chạy (với các giá trị là sang phải, xuống, sang trái, lên) và hướng chạy ban đầu là sang phải (hằng tượng trưng RIGHT). Hàm move sẽ xác định vị trí tiếp theo. Để trợ giúp việc chạy ta dùng thêm các biến l, r, t, b tương ứng với biên trái, phải, trên, dưới của vùng đang chạy.

Hàm move có thể được viết như sau.

Mã 4.3.12 – Hàm xác định vị trí tiếp theo

```
int move(int &i, int &j, int &dir,
         int &l, int &r, int &t, int &b) {
    while(l <= r && t <= b) {
        if(dir == RIGHT) {
            if(j < r) {
                j++;
                return 1;
            }
            else {
                dir = DOWN;
                t++;
            }
        }
        else if(dir == DOWN) {
            if(i < b) {
                i++;
                return 1;
            }
            else {
                dir = LEFT;
                r--;
            }
        }
        else if(dir == LEFT) {
            if(j > l) {
                j--;
                return 1;
            }
            else {
                dir = UP;
                b--;
            }
        }
        else { // dir == UP
            if(i > t) {
                i--;
                return 1;
            }
            else {
                dir = RIGHT;
```

```
                l++;
            }
        }
    }

    return 0;
}
```

Lưu ý, hàm `move` dựa trên thông tin hiện tại của vị trí (`i`, `j`), hướng chạy (`dir`), các biên (`l`, `r`, `t`, `b`) để xác định vị trí kế tiếp và có thể cập nhật hướng chạy mới hay các biên mới nên các tham số trên vừa là đầu vào, vừa là đầu ra nên ta để là các tham biến (tức là có dấu & đằng trước tên). Hàm này cũng trả về giá trị cho biết là có chạy qua vị trí mới được không (trả về 1) hay không chạy được nữa (trả về 0) khi ta đã chạy xong trên mảng hai chiều. Ở đây ta không chạy được nữa (nghĩa là đã xong) khi mà vùng chạy không còn hợp lệ nghĩa là `l` > `r` hoặc là `t` > `b`. Phủ định điều kiện này ta có điều kiện của vòng `while` (nghĩa là vẫn chạy).

Hàm `move` trông thì dài nhưng lại rất trực quan (theo cách chạy) chẳng hạn trường hợp chạy sang phải (`dir == RIGHT`): nếu chưa đụng biên phải (`j < r`) thì ta chạy qua ô kế tiếp chính là ô bên phải (`j++`) còn nếu không (đã đụng biên phải) thì ta điều chỉnh hướng chạy xuống dưới (`dir = DOWN`) và tăng biên trên (`t++`) do đã chạy xong dòng hiện tại. Các hằng tượng trưng `RIGHT`, `LEFT`, `DOWN`, `UP` được `#define` trước đó, chẳng hạn:

```
#define LEFT    1
#define RIGHT   2
#define DOWN    3
#define UP      4
```

Ta cũng có thể thay hàm nhập trên (nhập từ người dùng khá lâu) bằng hàm nhập nhanh gọn sau để kiểm thử.

Mã 4.3.13 – Hàm nhập nhanh cho mảng hai chiều

```
void nhap1(int a[][100], int m, int n) {
    int d = 1;
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            a[i][j] = d++;
}
```

Hàm này nhập các số nguyên liên tiếp, bắt đầu từ 1 vào mảng hai chiều (theo dòng). Bạn thêm hàm này vào chương trình và trong hàm `main` thay lời gọi hàm `nhap` bằng lời gọi hàm `nhap1` (Dòng 7 Mã 4.3.8). Sau đây là kết xuất cho trường hợp kích thước mảng là 4×5.

```
Nhap kich thuoc mang: 4 5↵
```

```
Mang truoc khi sap:
    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
   16   17   18   19   20
Mang sau khi sap:
   20   19   18   17   16
    7    6    5    4   15
    8    1    2    3   14
    9   10   11   12   13
```

Với các yêu cầu sắp xếp khác nhau, ta chỉ cần cài đặt hàm chép lại phù hợp (nghĩa là hàm `copyfrom` và `move`). Việc chép từ mảng một chiều sang mảng hai chiều ở trên trông phức tạp (do việc chạy theo đường xoắn ốc khá phức tạp), trong trường hợp khác thì sẽ đơn giản hơn. Chẳng hạn như khi ta muốn sắp giảm dần theo cột (hết cột 1 sang cột 2, ...) thì ta có hàm chép đơn giản như sau.

Mã 4.3.14 – Hàm chép dữ liệu từ mảng một chiều sang mảng hai chiều trường hợp sắp theo cột

```
void copyfrom(int a[][100], int m, int n, int x[]) {
    int p = 0, i = 0, j = 0;
    do {
        a[i][j] = x[p];
        p++;
    }
    while(move(i, j, m, n));
}
```

Và hàm `move` cũng rất đơn giản.

Mã 4.3.15 – Hàm xác định vị trí tiếp theo trường hợp sắp theo cột

```
int move(int &i, int &j, int m, int n) {
    if(i == m - 1 && j == n - 1)
        return 0;

    if(i == m - 1) {
        i = 0;
        j++;
    }
    else
        i++;

    return 1;
}
```

Ở đây do cách chạy đơn giản nên ta không cần duy trì các thông tin như hướng chạy hay vùng đang chạy như trường hợp chạy xoắn ốc, hơn nữa ta cũng có ít trường hợp phải xét hơn. Bạn chạy thử và ngẫm nghỉ để rõ nhé (mặc dù đơn giản, dễ hiểu hơn trường hợp xoắn ốc nhiều). Sau đây là kết xuất cho trường hợp kích thước mảng là 4×5.

```
Nhap kich thuoc mang: 4 5↵
Mang truoc khi sap:
    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
   16   17   18   19   20
Mang sau khi sap:
   20   16   12    8    4
   19   15   11    7    3
   18   14   10    6    2
   17   13    9    5    1
```

Bạn có thể thay đổi cách chép lại (sửa hàm `copyfrom` và `move`) để có các cách sắp xếp khác nhau không, chẳng hạn sắp tăng dần theo dòng từ dưới lên? sắp giảm dần theo đường chéo xuống? …

Tới đây, có hai điều quan trọng bạn cần nắm:

- Từ việc sắp cứng (tăng dần), bằng cách dùng khái niệm nghịch thế ta đã có cài đặt tổng quát, linh hoạt, dễ dàng thay đổi cho các cách sắp xếp khác nhau (bằng cách cụ thể khái niệm nghịch thế tức là cài đặt hàm nghịch thế). Như vậy bằng cách nới rộng hay nhìn xa hơn ta có thể tổng quát hóa đoạn mã, nó sẽ linh hoạt và dễ dàng tùy biến cho các trường hợp cụ thể khác nhau. Bù lại mã tổng quát thường không hiệu quả bằng (chậm hơn) và dài hơn (mặc dù rõ ràng, dễ hiểu hơn). Đây là trade off: tổng quát/ít hiệu quả với cụ thể/hiệu quả hơn.
- Từ việc sắp xếp tại chỗ (in-place) ta cho phép mình chép dữ liệu ra một vùng tạm dễ thao tác hơn (out-of-place) ta đã có cài đặt dễ dàng hơn. Bù lại ta sẽ tốn vùng nhớ hơn và các thao tác cũng dài hơn (mặc dù rõ ràng, dễ hiểu hơn). Đây là trade off giữa tốn tài nguyên vs xử lý dễ dàng (nhiều tiền chi tiêu sẽ dễ dàng hơn, ít tiền chi tiêu sẽ hiệu quả hơn).

BÀI TẬP

Bt 4.3.1 Trong bài học trên ta đã dùng kĩ thuật xử lý in-place để sắp xếp mảng một chiều các số nguyên sao cho: số chẵn đứng trước số lẻ, số chẵn giảm dần, số lẻ tăng dần. Hãy làm lại bằng kĩ thuật xử lý out-of-place. (*Gợi ý*: chép riêng các số chẵn và sắp giảm dần, chép riêng các số lẻ và sắp tăng dần, rồi nối theo thứ tự.)

Bt 4.3.2 Sắp xếp mảng một chiều các số nguyên sao cho "các số dương đứng trước các số 0, các số 0 đứng trước các số âm, các số dương tăng dần, các số âm giảm dần" bằng cả 2 kĩ thuật: in-place và out-of-place. So sánh 2 kĩ thuật trong trường hợp này.

Thật ra ta có một cách làm *ad hoc*[2] cho trường hợp này như sau: sắp giảm dần trên toàn bộ mảng (do đó các số dương sẽ đứng trước các số 0, các số 0 đứng trước các số âm) và hơn nữa các số âm giảm dần. Sau đó ta xác định phần đầu của mảng chứa các số dương (có thể xem là mảng con) và sắp tăng dần phần này. Bạn cũng hãy cài đặt cách làm này bằng kĩ thuật in-place (nghĩa là làm ngay trên mảng ban đầu mà không dùng thêm mảng phụ).

Bt 4.3.3 Làm yêu cầu tương tự Bài tập 4.3.2 cho việc sắp xếp tăng dần mảng một chiều các số nguyên không âm sao cho "số chia hết cho 3 đứng trước các số chia 3 dư 1 và các số chia 3 dư 1 đứng trước các số chia 3 dư 2".

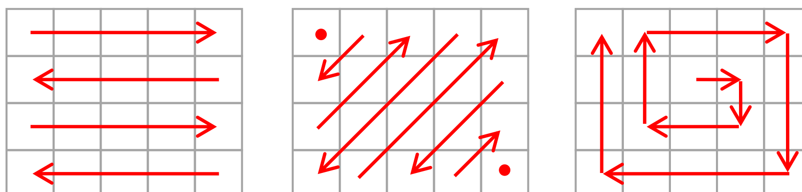
Bt 4.3.4 Một mảng các số nguyên được gọi là "cân bằng"[3] nếu nó không chứa số 0 và số lượng số âm bằng với số lượng số dương. Làm yêu cầu tương tự Bài tập 4.3.2 cho việc sắp xếp tăng dần một mảng "cân bằng" sao cho các số âm và dương xen kẽ nhau.

Bt 4.3.5 Xóa các số âm ra khỏi mảng một chiều các số nguyên (giữ đúng thứ tự tương đối của các số không âm) bằng cả 2 kĩ thuật: in-place và out-of-place. So sánh 2 kĩ thuật trong trường hợp này. (*Gợi ý*: trong trường hợp này thì in-place nghĩa là không được dùng thêm mảng phụ còn out-of-place là được dùng thêm mảng phụ.)

Bt 4.3.6 Làm yêu cầu tương tự Bài tập 4.3.5 cho việc xóa các số trùng nhau trong mảng (chỉ giữ lại số ở lần xuất hiện đầu tiên trong mảng).

Bt 4.3.7 Tương tự bài học trên, sắp xếp mảng hai chiều các số nguyên theo các yêu cầu như hình sau bằng cả 2 kĩ thuật in-place và out-of-place.

Hình 4.3.2 – Hình minh họa các yêu cầu sắp xếp của Bài tập 4.3.7



Mảng là công cụ vô giá của C!

[2] Ad hoc (từ Latin) nghĩa là giải pháp chuyên biệt cho một trường hợp cụ thể mà thường không dùng được cho các trường hợp khác và khó tổng quát hóa.
[3] Đây là thuật ngữ tôi chế ra, không liên quan tới bất kì khái niệm cân bằng nào.

BÀI 4.4

Bạn đã biết cách nhập vào một số nguyên? Theo thói quen thông thường, người dùng mô tả một số nguyên bằng cơ số 10[1]. Để nhập một số nguyên viết theo cơ số 10 bạn chỉ cần dùng hàm `scanf` với định dạng nhập `%d` (`d` là viết tắt của decimal – 10). Thế giả sử người dùng muốn mô tả số nguyên bằng cơ số 16 thì sao?[2] Đơn giản, dùng hàm `scanf` với định dạng nhập `%x` (`x` là viết tắt của hexa – 16). Ví dụ đoạn mã sau nhập một số nguyên theo cơ số 16 và xuất ra số đó theo cơ số 10:

```
int d;
printf("Nhap mot so (theo co so 16): ");
scanf("%x", &d);
printf("So da nhap (theo co so 10): %d\n", d);
```

Nhập chẳng hạn: `ff` (cơ số 16) thì số tương ứng là 255 (cơ số 10).

Thế giờ ta muốn nhập theo cơ số 2 thì sao? Không có `%b` đâu nha (`b` là viết tắt của binary). Ta viết hàm nhập vào một số nguyên không âm cơ số 2 như sau.

Mã 4.4.1 – Hàm nhập một số nguyên không âm viết theo cơ số 2

```
int nhap_coso2()
{
    char b[50];
    scanf("%s", b);

    int d = 0;
    for(int i = 0; b[i] != '\0'; i++)
        d = d * 2 + (b[i] - '0');

    return d;
}
```

Và có thể chạy thử bằng đoạn mã:

```
printf("Nhap mot so (theo co so 2): ");
int d = nhap_coso2();
printf("So da nhap (theo co so 10): %d\n", d);
```

[1] Số viết theo cơ số 10 còn gọi là số thập phân. Có lẽ thói quen dùng số thập phân là do con người ta có 10 ngón tay.

[2] Không phải do người ta có 16 ngón tay mà đó có lẽ là dân IT – cơ số 16 hay cơ số lũy thừa của 2 thì hợp với máy hơn. Xem lại Bài 1.7.

Lưu ý, ta chỉ xử lý trường hợp số không âm thôi, như vậy giả định của hàm này là người dùng nhập dãy các chữ số 0, 1 thôi. Hai dòng đầu tiên giúp ta nhập vào một *chuỗi*. Ta đã nghe nhiều về chuỗi, bài này sẽ chỉ rõ hơn nó là cái gì. Trước hết chuỗi là mảng (một chiều) các kí tự, tức là danh sách các kí tự. Ta dùng một mảng các kí tự `b` với sức chứa là 50 kí tự (điều đó cũng có nghĩa là có thêm giả định nữa là người dùng nhập không quá … 49 chữ số nhị phân?!). Sau đó ta cho nhập dãy chữ số như là chuỗi bằng định dạng nhập `%s`. Sau đó vòng lặp `for` cho phép ta xử lý chuỗi này để tính số nguyên tương ứng bỏ vào biến nguyên `d`. Sau vòng lặp, lệnh `return` sẽ trả về giá trị cho hàm là `d`.

Thật ra khi bạn gọi `scanf` với `%d` hay `%s` nó cũng đều làm tương tự nhau: nhập chuỗi và xử lý chuỗi để được số nguyên tương ứng. Từ *chuỗi chữ số* nhị phân làm sao tính được số nguyên mà nó biểu diễn, chẳng hạn từ chuỗi nhị phân: 1101 làm sao tính được số nguyên tương ứng là 13. Ta dùng qui tắc tính Horner[3]: xuất phát từ `d = 0` ta duyệt từ trái cho đến hết chuỗi, gấp đôi `d` và cộng thêm giá trị của chữ số. Như vậy ta cần duyệt từ đầu (trái) đến hết chuỗi, vòng lặp `for` với biến chạy `i` khởi động là 0 và tăng 1 mỗi lần chạy có thể làm điều đó. Nhưng quan trọng, ta làm sao biết đã hết chuỗi chưa hay điều kiện chạy của vòng lặp `for` này là gì? Với mảng thông thường (kí tự, số, …) ta có dùng kèm thông tin là số phần tử trong mảng (thường kí hiệu là biến nguyên `n`, do đó điều kiện chạy là `i < n`). Ở đây ta không có thông tin này. Trong trường hợp này, C dùng mẹo là dùng kí tự đặc biệt `'\0'`, gọi là kí tự `NULL`[4] để đánh dấu kết thúc chuỗi. Chẳng hạn bạn nhập `1101` thì các kí tự được bỏ vào `b` sẽ là `'1'`, `'1'`, `'0'`, `'1'` và `NULL`, tức là 5 kí tự được đặt vào mảng `b` (theo thứ tự tại vị trí 0, 1, 2, 3, 4). Đó chính là lí do mà ở trên người dùng không được nhập quá 49 chữ số vì phải dành chỗ cho kí tự `NULL`. Tất cả các kí tự khác đều có mã khác `NULL`. Vậy điều kiện chạy là kí tự tại vị trí `i` không phải là kí tự `NULL` `b[i] != '\0'` hay điều kiện dừng là `b[i] == '\0'`[5]. Như vậy, *chuỗi là mảng kí tự được đánh dấu kết thúc bởi kí tự* `NULL`.

Còn điều lưu ý rất bình thường nữa là: các phần tử của chuỗi là kí tự, mà bạn đã biết đó là mã ASCII của kí tự. Bạn phải phân biệt được kí tự `'1'` (mã ASCII là số nguyên 49) với giá trị 1 (số nguyên 1) và kí tự `'0'` với giá trị 0. Đó là lí do mà để có giá trị 1 từ chữ số `'1'` ta phải lấy `'1'` (số nguyên 49) trừ cho `'0'` (số nguyên 48).

---

[3] Tham khảo phần "Conversion to decimal" của bài viết "Binary number" trên trang Wikipedia (https://en.wikipedia.org/wiki/Binary_number).

[4] Kí tự này có mã ASCII là 0 (Xem lại Bài 1.7). `NULL` là hằng tượng trưng được `#define` là 0 trong `stdio.h`

[5] Bạn có thể viết là `b[i] == NULL` hay `b[i] == 0` đều được.

Bây chừ giả sử ngược lại ta muốn xuất số nguyên (không âm) ra theo biễu diễn cơ số 2 thì sao? Nếu xuất theo cơ số 10 hay 16 bạn có thể dùng `printf` với định dạng %d hay %x. Thực hiện thuật toán chia dư[6], bạn có thể viết như sau.

Mã 4.4.2 – Hàm xuất một số nguyên không âm viết theo cơ số 2 (phiên bản chưa đúng)

```
void xuat_coso2(int d)
{
    do
    {
        printf("%d", d % 2);
        d = d / 2;
    } while(d > 0);
}
```

Rất tiếc, chưa được! Bạn gọi `xuat_coso2(13)` để xuất ra biểu diễn nhị phân của số 13 thì kết quả là `1011`. Chưa đúng! Thật ra hàm trên đã làm đúng một nữa, trừ thứ tự các chữ số đã bị đảo ngược. Ta cần xuất ngược lại mới đúng. Điều này gợi ý rằng ta dùng mảng để chứa chuỗi các chữ số, dùng thuật toán như trên để có chuỗi chữ số, sau đó đảo ngược thứ tự các chữ số trong chuỗi ta sẽ có kết quả là chuỗi nhị phân. Như vậy ta có thể viết lại như sau.

Mã 4.4.3 – Hàm xuất một số nguyên không âm viết theo cơ số 2 (phiên bản đúng)

```
void xuat_coso2(int d)
{
    char b[50];

    int i = 0;
    do
    {
        b[i] = d % 2 + '0';
        d = d / 2;
        i = i + 1;
    } while(d > 0);
    b[i] = '\0';

    dao_chuoi(b);

    printf("%s", b);
}
```

[6] Tham khảo phần "Conversion from decimal" của bài viết "Binary number" trên trang Wikipedia (https://en.wikipedia.org/wiki/Binary_number).

Ta dùng mảng kí tự b để chứa chuỗi. Lưu ý Dòng 8 cộng thêm mã ASCII của kí tự `'0'` để có chữ số `'0'`, `'1'`, (chứ không phải giá trị 0, 1). Ta cần đặt kí tự `NULL` để kết thúc chuỗi (và do đó b là chuỗi chứ không chỉ là mảng kí tự) như Dòng 12. Sau đó ta gọi hàm `dao_chuoi` để đảo vị trí các chữ số trong chuỗi b trước khi xuất chuỗi bằng hàm `printf` với định dạng `%s`. Hàm đảo chuỗi có thể được viết như sau.

Mã 4.4.4 – Hàm đảo chuỗi

```
void dao_chuoi(char s[])
{
    int j = tinh_dodaichuoi(s) - 1;
    int i = 0;
    while(i < j)
    {
        char t = s[i];
        s[i] = s[j];
        s[j] = t;

        i++; j--;
    }
}
```

Ý tưởng để đảo chuỗi rất đơn giản. Ta đặt `i` ở biên trái, `j` ở biên phải của chuỗi. Hoán đổi kí tự tại vị trí `i` với kí tự tại vị trí `j`. Sau đó tăng `i` và giảm `j`. Ta lặp lại việc này khi `i` còn ở bên trái `j` (tức là `i < j`). Không có gì nhiều ngoại trừ việc biên trái của chuỗi chính là vị trí 0, còn biên phải của chuỗi là vị trí nào? Đó là vị trí của kí tự cuối cùng trong chuỗi (vị trí ngay trước vị trí kí tự `NULL`). Gọi độ dài (hay chiều dài) của chuỗi là số lượng kí tự trong chuỗi, chẳng hạn chuỗi `1101` có độ dài là 4, thì vị trí của kí tự cuối cùng trong chuỗi kém hơn độ dài 1 đơn vị. Vậy làm sao tính độ dài của chuỗi? Ta có thể viết hàm như sau.

Mã 4.4.5 – Hàm tính độ dài chuỗi

```
int tinh_dodaichuoi(char s[])
{
    int i = 0;
    while(s[i] != '\0')
        i++;

    return i;
}
```

Để tính độ dài chuỗi ta chỉ cần tìm vị trí của kí tự `NULL`. Để làm điều này ta duyệt từ đầu chuỗi đến khi đụng kí tự `NULL`, như vậy tham số `s` là chuỗi chứ không chỉ là mảng kí tự vì là chuỗi thì có kí tự kết thúc `NULL` còn mảng kí tự thì không nhất thiết.

Thật ra thì ta không cần tốn công sức để làm hết như vậy (mặc dù nó cũng không đáng kể). Chuỗi là dạng dữ liệu quan trọng trong C nên C đã cung cấp nhiều hàm để làm các thao tác/xử lý cơ bản trên chuỗi. Các hàm này nằm trong `string.h` và là một phần của thư viện chuẩn C. Chẳng hạn để tính độ dài chuỗi ta có thể dùng hàm `strlen` với công việc (và prototype) tương tự như hàm `tinh_dodaichuoi` mà ta đã tự mình viết lấy. Ta nên tự viết hay dùng cái có sẵn? Khi đang học thì bạn nên tự viết để hiểu rõ hơn, còn khi đã đi làm thì nên dùng cái có sẵn để đỡ tốn công sức hơn. Hiển nhiên là thư viện chỉ cung cấp các thao tác cơ bản thôi, những thao tác không có sẵn ta phải tự viết, chẳng hạn thao tác đảo chuỗi. Tuy nhiên khi tự viết thì ta cũng nên tận dụng cái đã có, chẳng hạn thao tác đảo chuỗi dùng thao tác tìm chiều dài chuỗi thì nên dùng hàm `strlen` có sẵn. Bạn hãy rèn luyện kĩ năng tra cứu bằng cách tìm hiểu thêm các hàm xử lý chuỗi trong thư viện chuẩn của C nhé.

MỞ RỘNG 4.4 – Dùng chuỗi biểu diễn số nguyên lớn

Đố bạn viết được chương trình cho nhập một số nguyên không âm và kiểm tra xem đó là số chẵn hay không? Có lẽ bạn sẽ viết như thế này:

Mã 4.4.6 – Đoạn mã "thông thường" kiểm tra số chẵn

```
unsigned int d;
printf("Nhap so nguyen khong am: ");
scanf("%u", &d);

if(d % 2 == 0)
    printf("So chan\n");
else
    printf("So le\n");
```

OK! Đoạn mã tốt. Đố bạn viết được chương trình cho nhập một số nguyên không âm và kiểm tra xem số đó có chia hết cho 3 hay không? Có lẽ bạn sẽ viết như thế này.

Mã 4.4.7 – Đoạn mã "thông thường" kiểm tra số chia hết cho 3

```
unsigned int d;
printf("Nhap so nguyen khong am: ");
scanf("%u", &d);

if(d % 3 == 0)
    printf("So chia het cho 3\n");
else
    printf("So khong chia het cho 3\n");
```

Uhm… đoạn mã không tốt! Có lẽ bạn không chấp nhận được chuyện này.

(Tại sao ở trên đúng mà dưới lại sai được. Chẵn tức là chia hết cho 2 thôi mà. Chia hết cho 2 thì được mà sao chia hết cho 3 thì lại có vấn đề. Mình cũng thử thấy tốt mà: 3333 chia hết cho 3 nè, 3333331 không chia hết cho 3 nè, 3000000000 cũng chia hết cho 3 nè, … Lầm bầm … Lầm bầm). Không phục đúng không? Bạn hãy nhập số 3 000 000 000 000[7]. Số này chia hết cho 3 (3 ngàn tỉ chia 3 được 1 ngàn tỉ không dư tí nào), tuy nhiên đoạn mã trên của bạn sẽ báo là không chia hết cho 3. Dù vậy đoạn mã kiểm tra chẵn vẫn báo đúng kết quả (là chẵn với số 3 ngàn tỉ).

Tại sao vậy? Có lẽ bạn đã biết được lí do rồi. Khi số nhập không quá lớn thì cả hai đoạn mã trên đều tốt. Nhưng khi số nhập quá lớn (lớn hơn 4 tỉ mấy[8]) thì sẽ xảy ra tràn số. Khi đó, d sẽ không chứa giá trị mà người dùng đã nhập mà là một số nguyên không âm khác[9]. Vậy tại sao kiểm tra chẵn (chia hết cho 2) vẫn hoạt động tốt trong khi kiểm tra chia hết cho 3 thì không? Lí do là khi tràn số thì d sẽ chứa số khác với số được nhập nhưng nó vẫn bảo toàn tính chẵn/lẻ (tức là số mà d chứa cũng sẽ chẵn/lẻ nếu số nhập là chẵn/lẻ), tuy nhiên nó không bảo toàn tính chia hết cho 3 (tức là nếu số nhập chia hết/không chia hết cho 3 thì số mà d chứa không chắc chia hết/không chia hết cho 3). Tại sao như vậy? Do cách nhập giá trị của hàm `scanf` mà tôi sẽ chỉ các bạn sau. Điều cũng dễ hình dung là trên máy thì 2 (hay lũy thừa của 2) luôn được ưu ái hơn các số khác.

Ta sẽ khắc phục để cho phép kiểm tra mà không sợ tràn số. Để làm như vậy ta sẽ dùng chuỗi để nhập số và xử lý chuỗi thay vì số. Mặc dù không cần nhưng sau đây là chương trình như vậy cho việc kiểm tra số chẵn.

Mã 4.4.8 – Đoạn mã "tốt hơn" kiểm tra số chẵn

```
char d[100];
printf("Nhap so nguyen khong am: ");
scanf("%s", d);

if((d[strlen(d) - 1] - '0') % 2 == 0)
    printf("So chan\n");
else
    printf("So le\n");
```

Ta đã thay d kiểu `unsigned int` thành kiểu chuỗi và cho nhập chuỗi thay vì số (`scanf` với `%s` thay vì `%u`). Khi đó d sẽ "chứa đầy đủ số" người dùng đã nhập mà không sợ bị tràn (không quá 99 chữ số nhé). Hiển nhiên khi

---

[7] Số 3 ngàn tỉ, tôi đã viết tách ra để bạn dễ nhìn nhưng khi nhập thì phải viết liền lại nhé.

[8] Coi lại phạm vi của số nguyên không dấu trong Bài 3.5?.

[9] Bạn có thể gọi `printf("%u", d)` để biết giá trị mà d thật sự chứa.

đó d là chuỗi số (tức là chuỗi các chữ số) chứ không phải là số. Tuy nhiên một số là chẵn khi chữ số cuối cùng (chữ số hàng đơn vị) là chẵn nên ta không cần chuyển d thành số, ta chỉ cần chuyển chữ số cuối cùng thành số và kiểm tra tính chẵn lẻ của nó. Xem điều kiện của lệnh `if` trên là đủ hiểu hén. (Hi vọng vậy!)

Thế kiểm tra chia hết cho 3 thì sao? Khó hơn chút xíu, ta dựa vào dấu hiệu chia hết cho 3: một số nguyên không âm là chia hết cho 3 khi và chỉ khi tổng các chữ số thập phân của nó chia hết cho 3. Để mã rõ ràng hơn, ta tách riêng hàm kiểm tra chia hết cho 3 và lệnh `if` trên thành:

```
if(chiahetcho3(d))
    printf("So chia het cho 3\n");
else
    printf("So khong chia het cho 3\n");
```

Và hàm `chiahetcho3` sẽ xử lý chuỗi số vào và trả về đúng (1) khi tổng các chữ số chia hết cho 3 và ngược lại thì trả về sai (0).

Mã 4.4.9 – Hàm kiểm tra chuỗi số thập phân có chia hết cho 3

```
int chiahetcho3(char d[])
{
    int s = 0;
    for(int i = 0; d[i] != '\0'; i++)
        s += d[i] - '0';
    if(s % 3 == 0)
        return 1;
    return 0;
}
```

Bạn thử lại xem nhé. 3 ngàn tỉ hay 3 tỉ ngàn tỉ ngàn tỉ đều đúng (miễn là không quá 99 chữ số, còn muốn nhiều hơn nữa thì nâng sức chứa của mảng d lên nhé).

Điều cần học hỏi ở đây là cách dùng chuỗi để lưu trữ và xử lý số giúp không tràn số, tức là làm việc với số siêu siêu lớn. Thực ra cách làm như vậy cũng có thể dùng không chỉ cho số nguyên mà số thực và mọi thứ có thể biểu diễn như là dãy kí hiệu nào đó, tức là có thể biểu diễn như là chuỗi. Bù lại cách làm này có nhược điểm là thường kém hiệu quả: tốn bộ nhớ lưu trữ và thời gian xử lý chậm. Nhưng cũng không hẳn, tùy bài toán, như kiểm tra số chẵn lẻ hay chia hết cho 3 ở trên vẫn rất hiệu quả đấy chứ.

BÀI TẬP

Bt 4.4.1 Làm lại Bài tập 1.5.6, Bài tập 1.5.7 và Bài tập 1.5.8 nhưng cho số nguyên không âm siêu lớn (không quá 100 chữ số thập phân). (Gợi ý: bạn biết rồi đó, đánh lừa thôi, làm trên chuỗi số chứ không phải là số.)

Bt 4.4.2 Nhập hai số nguyên không âm và xuất ra minh họa việc tính tay phép cộng hai số nguyên đó. Ví dụ như kết xuất sau:

```
Nhap a: 123456↵
Nhap b: 78910↵
    123456
   + 78910
   -------
    202366
```

Tương tự cho phép trừ. Đố bạn làm được cho phép nhân và phép chia đó.

Bt 4.4.3 Hàm `display` ở Bài 4.2 (Mã 4.2.3) xuất bộ đệm các kí tự ra màn hình. Hãy viết lại hàm này bằng cách xuất chuỗi. (*Gợi ý*: thêm kí tự kết thúc chuỗi vào cuối mỗi dòng kí tự để biến nó thành chuỗi.)

Bt 4.4.4 Tra cứu để hiểu khái niệm chuỗi "palindrome" và viết hàm C kiểm tra xem một chuỗi có là palindrome không.

Bt 4.4.5 Nhập một chuỗi và xuất ra bảng tần số của các kí tự xuất hiện trong chuỗi. Đặc biệt cho biết kí tự có tần số cao nhất. (Tần số là số lần xuất hiện.)

Bt 4.4.6 Tương tự Bài tập 4.4.5 nhưng cho cặp 2 kí tự liền kề (*bigram*), bộ 3 kí tự liền kề (*trigram*) hay tổng quát là *n-gram*[10].

Bt 4.4.7 Tra cứu để biết về chuỗi số La Mã (Roman numeral) và viết hàm `roman` để tính giá trị của một chuỗi số La Mã. Ví dụ: `roman("MDCLXVI")` trả về số nguyên 1666.

Bt 4.4.8 Một cách để bạn tăng công lực xử lý chuỗi rất nhanh là: tra cứu các hàm xử lý chuỗi/kí tự trong thư viện chuẩn của C và tự mình viết lại các hàm đó.

Bt 4.4.9 Vì chuỗi thực ra cũng là mảng (mảng kí tự được đánh dấu kết thúc bởi kí tự NULL) nên cũng có một kho tàng bài tập đồ sộ về chuỗi của C. Làm tương tự Bài tập 4.1.7 cho chuỗi. Hơn nữa có một khái niệm quan trọng trên chuỗi là *chuỗi con* (substring) mà có nhiều yêu cầu xử lý cho nó. Chẳng hạn: hãy tìm hiểu bài toán *tìm chuỗi con chung dài nhất* (longest common substring) và viết chương trình giải bài toán này.

Chuỗi là công cụ vô giá của C!

[10] https://en.wikipedia.org/wiki/N-gram

BÀI 4.5

Bạn hãy xem lại chương trình tìm số nguyên lớn nhất trong *n* số nguyên mà người dùng nhập ở Bài 4.1 (Mã 4.1.1). Giả sử ta muốn làm việc với số thực thay vì số nguyên (tức là tìm số thực lớn nhất trong *n* số thực mà người dùng nhập). Ta cần sửa chương trình ở các chỗ sau:

(1) `x`, `M` kiểu `int` sửa thành kiểu `double`.

(2) giá trị nhỏ nhất (kiểu `int`) là `INT_MIN` sửa thành số thực nhỏ nhất. Một giá trị đặc biệt như vậy là $-\infty$. Giá trị này có thể có được bằng tỉ số `1.0/0.0`.

(3) Nhập vào số thực `x` thay vì nhập số nguyên.

(4) Thực hiện phép so sánh số thực `x` với `M` thay vì phép so sánh số nguyên.

(5) Thực hiện phép gán thực `x` vào `M` thay vì gán số nguyên.

(6) Xuất ra số thực `M` thay vì số nguyên `M`.

Bạn thử sửa lại dựa vào gợi ý trên rồi chạy thử nhé. Sau đây là đáp án.

Mã 4.5.1 – Chương trình tìm số lớn nhất trong *n* số thực người dùng nhập

```c
#include <stdio.h>

int main()
{
    int n;
    printf("Ban muon nhap bao nhieu so: ");
    scanf("%d", &n);

    double x, M = -1.0/0.0;
    for(int i = 1; i <= n; i++)
    {
        printf("Nhap so thu %d: ", i);
        scanf("%lf", &x);
        if(x > M)
            M = x;
    }

    printf("So lon nhat la: ");
    printf("%lf", M);

    return 0;
}
```

Đặc biệt (4) và (5) không cần thay đổi gì. Các toán tử so sánh cùng kí hiệu cho các kiểu số (nguyên và thực các kiểu), C sẽ tự động gọi thao tác so sánh phù hợp (nguyên hay thực) dựa trên kiểu của toán hạng (`x`, `M`). Toán tử gán dùng chung cho mọi kiểu dữ liệu.

Bây chừ giả sử ta muốn làm việc với số hữu tỉ thì sao? Số hữu tỉ còn gọi là phân số, có dạng *a*/*b* với *a*, *b* là các số nguyên, *b* ≠ 0, *a* gọi là tử số, *b* gọi là mẫu số (xem lại Bài 3.1). Giống như trường hợp của số thực ta cần sửa 6 chỗ ở trên để có chương trình làm việc với phân số thay vì số nguyên.

Đầu tiên, ở (1) ta cần khai báo lại các biến `x`, `M` là kiểu phân số. Tuy nhiên ta không có từ khóa nào cho kiểu phân số như `int` hay `double`. Vấn đề là kiểu này không có sẵn trong C. Ta sẽ phải tự mình định nghĩa lấy kiểu này. Như đã nói, một phân số có thể xem là một cặp 2 số nguyên một là tử và một là mẫu (dấu / chỉ là kí hiệu hình thức để phân cách số đầu gọi là tử, số sau gọi là mẫu). Cặp số này luôn đi với nhau, gắn chặt với nhau không thể tách rời:) Những dạng dữ liệu (kiểu dữ liệu) như vậy có thể được khai báo bằng *kiểu cấu trúc* (*struct*) trong C và đây là chủ đề của bài này. Ở (2) ta cần tạo ra một hằng phân số là phân số nhỏ nhất. Phân số như vậy có mẫu là 1 và tử chính là số nguyên nhỏ nhất. Ở (3), (4), (5), (6) ta cần thao tác nhập, so sánh lớn hơn, gán và xuất phân số thay vì số nguyên. Tóm lại ta cần *định nghĩa kiểu* phân số gồm 2 thành phần là tử số và mẫu số đều có kiểu nguyên với các thao tác tương ứng ở trên. Để làm điều này ta cần *khai báo kiểu* phân số, *khai báo các thao tác* trên kiểu phân số và *cài đặt các thao tác* này. Để tổ chức chương trình tốt (xem lại Bài 3.7) ta sẽ tạo một module để định nghĩa kiểu phân số. Module này gồm file tiêu đề `phanso.h` chứa khai báo kiểu (cùng với khai báo các thao tác) và `phanso.cpp` chứa cài đặt các thao táo (các hàm).

File tiêu đề `phanso.h` được viết như sau.

Mã 4.5.2 – File tiêu đề `phanso.h` của module định nghĩa kiểu Phân Số

```
1  struct PhanSo
2  {
3      int tu;
4      int mau;    // khac khong
5  };
6
7  PhanSo int2PhanSo(int d);
8  void nhap(PhanSo &p);
9  void xuat(PhanSo p);
10 int lonhon(PhanSo x, PhanSo y);
11 void ganPhanSo(PhanSo &lh, PhanSo rh);
```

Các dòng từ Dòng 1 đến Dòng 5 khai báo kiểu Phân Số. Từ khóa `struct` được dùng để báo đây là kiểu cấu trúc tức là *kiểu phức hợp*, gồm nhiều *thành phần*. `PhanSo` là *tên kiểu*. Khác với `int`, `double` là các từ khóa. Tên

kiểu cấu trúc là định danh và tùy ý ta đặt (dĩ nhiên là theo qui định của định danh). Tên kiểu được dùng để chỉ đến kiểu nên ta đặt sao cho gợi nhớ, ở đây nên là `PhanSo` (Phân Số) hoặc `Rational` (số hữu tỉ). Bên trong cặp ngoặc nhọn[1] là *khai báo các thành phần*. Mỗi thành phần gồm *kiểu thành phần*, *tên thành phần*, kết thúc bởi dấu `;` (cú pháp như khai báo biến). Ở đây kiểu `PhanSo` có hai thành phần tên là `tu` và `mau` đều có kiểu `int` tương ứng với tử số và mẫu số của phân số. Ngoài ra kiểu `PhanSo` có giả định là thành phần `mau` (mẫu) luôn khác không. Lưu ý rằng, trong các trường hợp đơn giản thì kiểu của các thành phần là kiểu cơ bản (số nguyên, số thực, kí tự, luận lý, xem lại Bài 3.5), trường hợp phức tạp hơn, chúng có thể là mảng, chuỗi hay thậm chí là kiểu cấu trúc khác.

Sau khai báo kiểu là prototype của các hàm cung cấp các thao tác tương ứng trên kiểu phân số. Không có gì mới, ngoài chuyện tên kiểu phân số (`PhanSo`) tham gia vào hệ thống *kiểu có sẵn* của C (`int`, `double`, ...). Kiểu phân số còn được gọi là *kiểu do người dùng định nghĩa*. Khi ta đã định nghĩa kiểu này thì ta được dùng nó như bao kiểu khác như các kiểu đã có sẵn như `int`, `double`. Cũng có sự phân biệt đối xử đó là với các kiểu số thì C dùng các toán tử để kí hiệu cho các thao tác phổ biến (như toán tử > cho thao tác so sánh lớn hơn hay các hằng số cho các dữ liệu xác định) và các hàm nhập xuất chuẩn có hỗ trợ nó (`printf`, `scanf` với định dạng `%d`, `%lf`). Ở đây các thao tác đều viết thành các hàm (và như vậy có vẻ hơi cồng kềnh, không đẹp và kém tự nhiên).

Cài đặt của các hàm này đều để ở file cài đặt `phanso.cpp`.

Mã 4.5.3 – File cài đặt `phanso.cpp` của module định nghĩa kiểu Phân Số

```
#include <stdio.h>
#include "phanso.h"

PhanSo int2PhanSo(int d)
{
    PhanSo p;
    p.tu = d;
    p.mau = 1;

    return p;
}

void nhap(PhanSo &p)
{
    scanf("%d/%d", &p.tu, &p.mau);
}
```

[1] Nhớ rằng có dấu `;` sau dấu đóng ngoặc nhọn } như Dòng 7 cho thấy. Nếu bạn nào thấy khó chịu thì đây là qui định của C thôi, cũng tốt, nó nhắc ta rằng đó không phải là khối lệnh.

```
17
18 void xuat(PhanSo p)
19 {
20     if(p.mau == 1)
21         printf("%d", p.tu);
22     else
23         printf("%d/%d", p.tu, p.mau);
24 }
25
26 int lonhon(PhanSo x, PhanSo y)
27 {
28     int u = x.tu * y.mau - x.mau * y.tu;
29     int v = x.mau * y.mau;
30
31     if(u*v > 0)
32         return 1;
33     return 0;
34 }
35
36 void ganPhanSo(PhanSo &x, PhanSo y)
37 {
38     x.tu = y.tu;
39     x.mau = y.mau;
40 }
```

Hàm `int2PhanSo` (Dòng 4) tạo phân số có giá trị bằng với số nguyên `d`. Hàm này nhận vào số nguyên `d` và trả về phân số có tử là `d`, mẫu là `1`. Muốn vậy ta khai báo biến phân số `p` (Dòng 6), gán tử mẫu phù hợp cho `p` và trả về `p`. Việc khai báo biến kiểu cấu trúc giống như khai báo biến thông thường: tên kiểu `PhanSo` sau đó là tên biến `p`. Để truy cập đến tử của phân số `p` ta dùng *toán tử truy cập thành phần* (toán tử `.`) với cú pháp:

<tên biến (kiểu cấu trúc)>.<tên thành phần>

Đây là một biểu thức với toán tử `.` hai ngôi. Về ý nghĩa thì biểu thức này giúp ta chỉ đến thành phần tương ứng của dữ liệu trong biến cấu trúc. Do đó biểu thức này là l-value với kiểu là kiểu của thành phần đang được chỉ đến. Lệnh gán ở Dòng 7 giúp ta gán một giá trị `int` (số nguyên `d`) vào một l-value kiểu `int` (thành phần tử của phân số `p`). Tương tự, ở Dòng 8 ta truy cập đến `mau` (mẫu) của phân số `p` để bỏ giá trị nguyên `1` vào đó. Cuối cùng ta trả về phân số `p` đã được gán giá trị phù hợp ở Dòng 10. Kiểu trả về của hàm này, do đó là `PhanSo`.

Hàm `nhap` (Dòng 13) yêu cầu người dùng nhập vào một phân số với định dạng nhập là tử/mẫu. Hàm này cần một tham số vào kiểu `PhanSo` để chứa

phân số người dùng nhập[2]. Để nhập phân số, người dùng nhập vào 2 số nguyên phân cách bằng dấu `/`. Số nguyên thứ nhất được đặt vào tử của `p` và số nguyên thứ 2 được đặt vào mẫu của `p`. Ta truy cập vào `tu`, `mau` bằng toán tử `.`, đó là l-value cho nên có thể chứa giá trị nhập vào và ta cũng cần toán tử & đằng trước như khi ta nhập vào bao số nguyên (hay thực) khác.

Hàm `xuat` (Dòng 18) xuất phân số `p` ra màn hình. Để xuất đẹp, ta kiểm tra mẫu số, nếu là 1 thì ta chỉ xuất ra tử thôi (Dòng 21), nếu không thì ta xuất đầy đủ tử và mẫu (Dòng 23). Lưu ý `p` là tham số vào và `p.mau` là một số nguyên nên ta có thể dùng toán tử == để kiểm tra bằng (số nguyên).

Hàm `lonhon` (Dòng 26) kiểm tra phân số `x` có lớn hơn phân số `y` hay không, do đó trả về giá trị luận lý: đúng nếu `x` lớn hơn `y`, và sai nếu `x` không lớn hơn `y`. Như thường lệ ta sẽ dùng giá trị nguyên thay cho giá trị luận lý với 1 là đúng và 0 là sai. Để so sánh hai phân số nhớ là: $a/b > c/d$ khi và chỉ khi $a/b - c/d > 0$ khi và chỉ khi $(ad - bc)/(bd) > 0$ nghĩa là $(ad - bc)$ và (bd) cùng dấu (cùng âm hoặc cùng dương) nghĩa là $(ad - bc)*(bd) > 0$. Mặc dù cách làm trên chứng tỏ bạn giỏi Toán nhưng đó không là cách làm tốt do khả năng tràn số rất cao (phép nhân hai số nguyên rất dễ gây tràn). Tốt hơn là $a/b > c/d$ khi $a/b > c/d$!:) Không hiểu đúng không. Đọc cách viết đúng sau sẽ hiểu.

Mã 4.5.4 – Hàm so sánh `lonhon` được viết lại tốt hơn

```
int lonhon(PhanSo x, PhanSo y)
{
    if(x.tu/(double)x.mau > y.tu/(double)y.mau)
        return 1;

    return 0;
}
```

Nói nôm na: ta đã chuyển phân số `x`, `y` thành các giá trị thực tương ứng và dùng phép so sánh lớn hơn đã có trên kiểu `double` (toán tử >). Bạn hãy thay cài đặt tốt hơn này vào mã trên trong file `phanso.cpp` nhé.

Hàm `ganPhanSo` (Dòng 36) gán giá trị của phân số `y` vào phân số `x` (`y` đóng vai trò bên phải còn `x` đóng vai trò bên trái phép gán). Lưu ý là `x` là tham số ra và `y` là tham số vào (`x` thay đổi giá trị theo giá trị của `y`) nên khai báo `x` có dấu & đằng trước. Để gán phân số `y` vào `x` ta gán tương ứng tử, mẫu của `y` vào tử, mẫu của `x`.

Sau khi đã có định nghĩa kiểu phân số (bao gồm khai báo kiểu, khai báo và cài đặt các thao tác trên kiểu) ta có thể viết chương trình tìm số lớn nhất trong các phân số đã nhập như sau. (Viết chương trình vào một file riêng nhé, chẳng hạn file `main.cpp` hay `program.cpp`).

---

[2] giống như `scanf` cho số nguyên cần biến nguyên (l-value) để chứa số nguyên người dùng nhập.

Mã 4.5.5 – Chương trình tìm phân số lớn nhất trong *n* phân số

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include "phanso.h"
4  int main()
5  {
6      int n;
7      printf("Ban muon nhap bao nhieu so: ");
8      scanf("%d", &n);
9  
10     PhanSo x, M = int2PhanSo(INT_MIN);
11     for(int i = 1; i <= n; i++)
12     {
13         printf("Nhap so thu %d: ", i);
14         nhap(x);
15         if(lonhon(x, M))
16             ganPhanSo(M, x);
17     }
18 
19     printf("So lon nhat la: ");
20     xuat(M);
21 
22     return 0;
23 }
```

Thật ra thì ta không cần cung cấp thao tác gán cho kiểu PhanSo (hàm ganPhanSo). Toán tử gán của C (toán tử =) dùng được cho mọi kiểu dữ liệu nên hiếm khi ta phải tự viết hàm để thực hiện phép gán cho kiểu nào đó[3]. Với kiểu cấu trúc thì toán tử gán sẽ thực hiện việc gán cho từng thành phần (tương tự như hàm ganPhanSo mà ta đã viết). Như vậy ở Dòng 16 trên ta có thay lời gọi đến hàm ganPhanSo bằng lệnh gán `M = x;`. Bấy giờ ta cũng nên bỏ thao tác gán khỏi kiểu PhanSo vì ta không dùng đến. Để làm điều đó, ta xóa khai báo hàm ganPhanSo trong file tiêu đề `phanso.h` (Dòng 13 Mã 4.5.2) và xóa cài đặt hàm ganPhanSo trong file mã `phanso.cpp` (Dòng 36 đến Dòng 40 Mã 4.5.3).

MỞ RỘNG 4.5 – Tràn phân số

[3] Ta sẽ thấy, cũng có tình huống mà ta phải tự viết hàm thực hiện phép gán cho kiểu chứ không dùng toán tử gán được.

Giả sử ta trang bị thêm cho kiểu PhanSo trên hai thao tác nữa là tạo phân số từ tử số, mẫu số và nhân hai phân số như sau.

Mã 4.5.6 – Thao tác tạo phân số từ tử số, mẫu số và nhân hai phân số

```
PhanSo taoPhanSo(int tu, int mau)
{
    PhanSo p;
    p.tu = tu;
    p.mau = mau;

    return p;
}

PhanSo nhanPhanSo(PhanSo x, PhanSo y)
{
    PhanSo p;
    p.tu = x.tu * y.tu;
    p.mau = x.mau * y.mau;

    return p;
}
```

Khi đó để tính tích của phân số 2/3 với 5/7 ta có thể dùng đoạn mã sau:

```
PhanSo x = taoPhanSo(2, 3);
PhanSo y = taoPhanSo(5, 7);
xuat(nhanPhanSo(x, y));
```

sẽ được kết quả là 10/21. Dễ thấy là thao tác nhân phân số trên làm đúng: a/b × c/d = (ac)/(bd). Tuy nhiên đời không đơn giản như vậy!:) Bạn dĩ nhiên biết rằng:

$$\frac{2}{1}\times\frac{3}{2}\times\ldots\times\frac{n-1}{n-2}\times\frac{n}{n-1}=\frac{2\times3\times\ldots\times(n-1)\times n}{1\times2\times\ldots\times(n-2)\times(n-1)}=\frac{n}{1}=n$$

với mọi số nguyên $n > 1$. Đoạn mã sau sẽ tính tích của dãy phân số trên

Mã 4.5.7 – Đoạn mã tính tích của dãy phân số

```
int n = 10;
PhanSo p = taoPhanSo(1, 1);
for(int i = 2; i <= n; i++)
    p = nhanPhanSo(p, taoPhanSo(i, i - 1));
xuat(p);
```

bằng cách đặt giá trị mong muốn cho n. Chẳng hạn với n là 10 như trên thì ta được kết quả là: 3628800/362880. Kết quả này không đẹp nhưng vẫn đúng (3628800/362880 = 10). Tuy nhiên với n lớn hơn một chút thì kết quả không còn đúng nữa, chẳng hạn với n là 15 thì kết quả là:

2004310016/1278945280 ≈ 1.57 ≠ 15. Hiện tượng này được (tôi) gọi là "tràn phân số" tương tự như tràn số nguyên (Mở rộng 1.2) hay tràn số thực (Mở rộng 1.3) mà ta đã biết. Dễ thấy rằng ta bị tràn phân số vì ta đã không *tối giản phân số*. Chẳng hạn 2/3 × 3/4 được 6/12 nhưng tốt hơn thì phải là 1/2 vì 1/2 = 6/12 nhưng 1/2 đơn giản hơn 6/12. Ta nói rằng 1/2 là phân số tối giản của 6/12. Làm sao để tối giản một phân số? Ta chỉ cần chia tử và mẫu cho *ước chung lớn nhất* của chúng. Chẳng hạn 6 và 12 có ước chung lớn nhất là 6 nên 6/12 có phân số tối giản là 1/2.

Làm sao để tìm ước chung lớn nhất của hai số nguyên? Euclid[4] đưa ra một thuật toán rất đơn giản và hiệu quả là thuật toán của Euclid (Euclid's algorithm[5]). Thuật toán này dựa vào 2 nhận xét rất đơn giản sau, giả sử *a, b* là các số nguyên không âm:

- Nếu *b* là 0 thì *a* là ước chung lớn nhất của *a* và *b*.
- Nếu không, gọi *r* là số dư khi chia *a* cho *b* thì ước chung lớn nhất của *b* và *r* chính là ước chung lớn nhất của *a* và *b*.

Từ đó ta dễ dàng viết hàm `ucln`[6] tìm ước chung lớn nhất như sau:

Mã 4.5.8 – Hàm cài đặt thuật toán Euclid tìm ước chung lớn nhất

```
// a, b khong am
int ucln(int a, int b)
{
    while(b != 0)
    {
        int r = a % b;
        a = b;
        b = r;
    }

    return a;
}
```

Khi đã có hàm tìm ước chung lớn nhất trên ta sửa lại hàm `taoPhanSo` và `nhanPhanSo` bằng cách tối giản phân số như sau với `abs` là hàm tính trị tuyệt đối của một số nguyên.

Mã 4.5.9 – Hàm tạo và nhân phân số có tối giản phân số

```
PhanSo taoPhanSo(int tu, int mau)
{
    PhanSo p;
```

[4] Nhà toán học Hy Lạp cổ đại.

[5] Tham khảo thuật toán Euclid ở trang Wikipedia (https://en.wikipedia.org/wiki/Euclidean_algorithm).

[6] Viết tắt của "ước chung lớn nhất". Từ viết tắt chuẩn hay dùng là gcd (greatest common divisor), tuy nhiên ở đây ta lỡ dùng tiếng Việt rồi:)

```
    int uc = ucln(abs(tu), abs(mau));
    p.tu = tu / uc;
    p.mau = mau / uc;

    return p;
}

PhanSo nhanPhanSo(PhanSo x, PhanSo y)
{
    return taoPhanSo(x.tu * y.tu, x.mau * y.mau);
}
```

Chạy lại Mã 4.5.7 với n là 15 ta được kết quả: 15. Không chỉ đúng mà còn đẹp nữa! Thậm chí với n là 10000 ta vẫn được kết quả đúng. Tuy nhiên như vậy không có nghĩa là ta đã giải quyết được vấn đề tràn phân số. Với n là 100000 thì kết quả không còn đúng nữa. Tại sao? Bạn có biết n tối đa là bao nhiêu để không bị tràn (nghĩa là vẫn được kết quả đúng) không?

BÀI TẬP

Trong bài học trên, ta đã viết module Phân Số với file tiêu đề `phanso.h` (Mã 4.5.2) và file cài đặt `phanso.cpp` (Mã 4.5.3) để định nghĩa kiểu `PhanSo` và viết chương trình dùng kiểu `PhanSo` để tìm phân số lớn nhất trong file `program.cpp` (Mã 4.5.5). Tương tự, trong các bài tập dưới đây, hãy viết module định nghĩa kiểu cấu trúc với các thành phần[7] và thao tác phù hợp cho dạng (kiểu) dữ liệu được yêu cầu[8] và viết chương trình minh họa dùng kiểu cấu trúc đó.

Bt 4.5.1 Phân số. (Hoàn chỉnh kiểu `PhanSo` trong bài học với đầy đủ các thao tác, xem Bài tập 3.1.4.)

Bt 4.5.2 Thời điểm trong ngày. (Xem Bài tập 2.4.6, Bài tập 3.1.5.)

Bt 4.5.3 Đa thức. (Xem Bài tập 4.1.1.)

Bt 4.5.4 Số nguyên siêu lớn. (Xem Bài tập 4.4.1, Bài tập 4.4.2.)

Bt 4.5.5 Số phức. (Tra cứu để hiểu rõ về số phức (complex number). *Gợi ý*: số phức với số thực giống như phân số với số nguyên.)

Bt 4.5.6 Vector các số thực. (Tra cứu để hiểu rõ về vector các số thực (euclidean vector). *Gợi ý*: dùng mảng một chiều.) Vector các phân số[9] hay số phức thì sao?

---

[7] Nhắc lại, các thành phần có thể có kiểu đơn giản nhưng cũng có thể là mảng, chuỗi hay kiểu cấu trúc khác.

[8] Có thể có nhiều file cài đặt nếu kiểu phức tạp có nhiều thao tác.

[9] Trong ngữ cảnh này thì nên gọi phân số là số hữu tỉ.

Bt 4.5.7 Ma trận các số thực. (Tra cứu để hiểu rõ về ma trận (matrix). *Gợi ý*: dùng mảng hai chiều.) Ma trận các phân số hay số phức thì sao?

Bt 4.5.8 Ta đã làm nhiều với mảng (một chiều, hai chiều) các số nguyên, số thực, kí tự. Hơn như vậy, C cho phép mảng chứa các phần tử (cùng) kiểu cấu trúc. Sau khi hoàn thành Bài tập 4.5.1, hãy viết chương trình cho nhập một mảng (một chiều, hai chiều) các phân số, xắp sếp theo vài cách khác nhau và xuất ra kết quả. Tương tự với Bài tập 4.5.2, Bài tập 4.5.4 và Bài tập 4.5.5.

Cấu trúc là công cụ vô giá của C!

BÀI 4.6

Ta đã biết thế nào là giai thừa của một số nguyên dương *n*:

$n! = 1 \times 2 \times \ldots \times (n-1) \times n$.

Ta cũng đã biết cách viết mã C để tính giai thừa theo công thức trên trong Bài 1.5 (Mã 1.5.4). Bài này là về một cách nhìn khác và do đó dẫn đến một cách cài đặt khác. Đó là *đệ qui*. Một ít về Toán và tư duy trừu tượng ta có:

$n! = 1 \times 2 \times \ldots \times (n-1) \times n$
$= (1 \times 2 \times \ldots \times (n-1)) \times n$ (tính kết hợp của phép nhân)
$= (n-1)! \times n$ (nhận diện khuôn mẫu)

Điểm khác biệt với cách nhìn mới là ta có cách tính *n*! rất ngắn gọn nếu biết (*n* – 1)!. Cụ thể, lấy giai thừa của (*n* – 1) nhân với *n* sẽ được giai thừa của *n*. Hơn nữa, một cách tầm thường, ta có 1! = 1. Như vậy ta có định nghĩa mới (dẫn đến cách tính mới) cho *n*! như sau:

- 1! = 1.
- $n! = (n-1)! \times n$ nếu $n > 1$.

Định nghĩa trên được gọi là *định nghĩa đệ qui*. Dòng đầu được gọi là *trường hợp cơ sở*, dòng thứ hai được gọi là *trường hợp đệ qui*. Định nghĩa này cũng dẫn đến cách tính (cách làm) đệ qui. Giả sử *fact*(*n*) là hàm tính giai thừa của *n* thì ta có thể tính *fact* như sau:

- *fact*(*n*) = 1 nếu *n* = 1.
- *fact*(*n*) = *fact*(*n* – 1) × *n* nếu *n* > 1.

Ta thử tính 4! theo cách này. Để tính 4! ta thấy 4 > 1 nên theo dòng thứ hai ta tính 3! rồi nhân với 4. Tương tự để tính 3! ta tính 2! rồi nhân với 3 và để tính 2! Ta tính 1! rồi nhân với 1. Bây chừ để tính 1! thì theo dòng 1 ta có 1! là 1. Đã có 1! là 1 ta nhân với 2 được 2! là 2. Đã có 2! là 2 ta nhân với 3 để có 3! là 6. Đã có 3! là 6 ta nhân với 4 để có 4! là 24. Vậy 4! là 24. Ta thấy như vậy việc thực thi đệ qui có vẻ rắc rối hơn việc thực thi cách làm thông thường nhưng cách mô tả (định nghĩa) thì lại rất đơn giản. Quan trọng là việc mô tả (viết mã) là việc của ta (lập trình viên) còn việc thực thi là việc của môi trường thực thi. Ta có thể viết mã ngắn gọn cho hàm tính giai thừa trên trong C như sau.

Mã 4.6.1 – Hàm đệ qui tính giai thừa

```
int fact(int n)
{
    if(n == 1)
        return 1;

    return fact(n - 1) * n;
}
```

Đây hầu như là viết lại nguyên si mô tả của cách tính *fact* ở trên. Cũng lưu ý là ta không cần điều kiện cho n > 1 (xem lại `return` trong Bài 3.4). Hàm `fact` trên được gọi là *hàm đệ qui* mà dấu hiệu dễ thấy là trong thân của nó có lời gọi đến chính nó (dĩ nhiên là với đối số khác). Ta có thể tính n! bằng cách gọi hàm trên, chẳng hạn:

```
int n = 4;
printf("%d! = %d\n", n, fact(n));
```

Bạn có thể cho người dùng nhập thử các giá trị n khác nhau để xem kết quả. Lưu ý là nhập giá trị nhỏ cho n thôi do hàm giai thừa lớn rất nhanh nên với n vừa phải thì n! đã rất lớn nên dẫn đến tràn số nguyên và kết quả không còn đúng nữa (xem Bài 3.5).

Để có thể làm đệ qui một bài toán nào đó ta phải xác định một (hoặc một vài) tham số của bài toán mà ta sẽ đệ qui theo đó. Hơn nữa tham số đó phải đệ qui được. Ở mức nhập môn này thì các *tham số đệ qui* đều có kiểu nguyên (các kiểu dữ liệu có thể đệ qui khác sẽ được học ở tầng cao hơn). Chẳng hạn ở bài toán tính giai thừa thì ta đệ qui theo giá trị cần tính là n. Tham số này có kiểu nguyên. Tiếp theo ta cần phát hiện cách tính lời giải cho bài toán từ lời giải của chính bài toán đó nhưng với tham số đệ qui "*nhỏ hơn*" (với kiểu số nguyên thì "nhỏ hơn" chính là nhỏ hơn <). Chẳng hạn ta phát hiện lời giải của n! chính là lời giải của (n – 1)! (cũng là giai thừa nhưng với tham số nhỏ hơn, $n - 1 < n$) nhân với n. Sau cùng ta phải thấy được lời giải của bài toán trong trường hợp tham số đơn giản (với kiểu số nguyên thì đó là các số nhỏ như 0, 1). Chẳng hạn ta thấy 1! là 1. Lời giải trong trường hợp này thường là tầm thường (dễ thấy) nhưng vai trò của nó là cực lớn: nếu không có nó, đệ qui sẽ không dừng được.

Hãy xét một bài toán khác, tính tổng các số từ 0 đến n, cụ thể với n là số nguyên không âm ta tính $S(n) = 0 + 1 + 2 + ... + n$. Ta có thể tính tổng này bằng một vòng `for` như khi tính giai thừa. Tuy nhiên ta thử làm bằng đệ qui. Trước hết ta cần xác định tham số đệ qui: bài toán có 1 tham số duy nhất, n. Nó có kiểu số nguyên nên ta sẽ đệ qui theo nó (chứ biết sao bây chừ!). Tiếp theo ta cần xác định qui tắc đệ qui:

$$S(n) = 0 + 1 + ... + n = (0 + 1 + ... + n - 1) + n = S(n - 1) + n.$$

để tính tổng tới n ta tính tổng tới n - 1 (cũng là tính tổng nhưng với tham số nhỏ hơn, $n - 1 < n$) rồi cộng thêm n. Sau cùng ta cần xác định lời giải trong

trường hợp tham số đơn giản, dễ thấy $S(0) = 0$. Vậy ta có thể viết hàm này trong C như sau.

Mã 4.6.2 – Hàm đệ qui tính tổng 0 + 1 + 2 + … + n

```
int S(int n)
{
    if(n == 0)
        return 0;

    return S(n - 1) + n;
}
```

Ta thấy khuôn mẫu hàm này rất giống hàm `fact` trên. Thật vậy các hàm đệ qui đều có khuôn mẫu rất giống nhau.

Thật ra thì để tính tổng các số tới n thì ta dùng công thức $S(n) = n(n+1)/2$ là nhanh nhất (tốn một phép cộng, một phép nhân và một phép chia nguyên) còn nếu cài bằng lặp thì ta sẽ tốn n phép toán, với cài đặt đệ qui thì số phép toán sẽ còn nhiều hơn (cũng tỉ lệ với n, $2n$ chẳng hạn)[1]. Ngoài lề một chút, có thể bạn tò mò là có một công thức tương tự như vậy cho giai thừa hay không. Những công thức dạng này (tốn cố định và thường ít phép toán) gọi là *công thức đóng*. Rất tiếc là không có công thức đóng nào cho giai thừa[2].

Cách tính giai thừa bằng cách dùng các vòng lặp (vòng for như ở Mã 1.5.4) được gọi là làm bằng *kĩ thuật lặp* còn cách tính bằng đệ qui (như ở Mã 4.6.1) được gọi là làm bằng *kĩ thuật đệ qui*. Mối quan hệ giữa đệ qui và lặp gắn bó mật thiết hơn là bạn có thể hiểu hay tưởng tượng bây giờ. Mọi dạng lặp có thể được chuyển hóa thành đệ qui và mọi dạng đệ qui đều ẩn tàng bên dưới là lặp[3]. Bạn hãy xem lại chương trình "Valentine ít tốn công" ở Bài 1.5 (Mã 1.5.3) là điển hình của kĩ thuật lặp. Nay ta viết lại nó bằng kĩ thuật đệ qui như sau.

Mã 4.6.3 – Mã 1.5.3 viết lại bằng kĩ thuật đệ qui[4]

```
#include <stdio.h>

void loveyou(int n)
{
    if(n > 0)
    {
```

[1] Điều này cũng cho thấy sức mạnh của Toán. Học Toán nhé!

[2] Nhưng ta có thể xấp xỉ giai thừa bằng công thức đóng.

[3] Nghe như bí quyết của XYZ chân kinh mà bạn chỉ có thể ngộ ra khi luyện đến tầng cao nhất.

[4] Hàm `loop` trong Mã 3.6.2 (Bài tập 3.6.5) cũng là hàm đệ qui với hoạt động tương tự.

```
        printf("I love you!\n");
        loveyou(n - 1);
    }
}

int main()
{
    loveyou(1000);

    return 0;
}
```

Ta đã viết hàm đệ qui `loveyou` mà khi được gọi thì nó thực thi như vòng lặp `for` của Mã 1.5.3. Bạn có biết trường hợp cơ sở (trường hợp dừng) của hàm đệ qui này là gì không? Đúng hơn thì hàm đệ qui trên tương đương với đoạn mã:

```
for(int i = n; i > 0; i = i - 1)
    printf("I love you!\n");
```

Ở đây ta cũng thấy cách làm đệ qui không dẫn đến cách làm hiệu quả hơn (ít phép toán và do đó chạy nhanh hơn) mà đệ qui lại thường kém hiệu quả hơn (nhiều phép toán, và tốn thời gian gọi hàm nhiều lần). Đệ qui, tuy nhiên, mang lại một cách nhìn (cách hiểu, cách nghĩ) đơn giản, dễ hiểu hơn và dẫn đến việc viết mã cũng đơn giản và mã cũng dễ hiểu hơn. Ta sẽ còn phân tích ưu điểm và khuyết điểm của đệ qui sau.

Như đã nói, ở tầng này ta đệ qui theo một tham số nguyên, tuy nhiên bản chất của tham số đó là gì thì cũng rất đa dạng. Ở hai bài toán trên ta *đệ qui theo độ lớn* của tham số *n*. Xét bài toán tính tổng các chữ số (theo cơ số 10) của một số nguyên không âm *n*, *sumdigit*(*n*). Ví dụ số 1234 có tổng các chữ số là 10 (1 + 2 + 3 + 4) tức *sumdigit*(1234) có giá trị là 10. Ở đây ta nhận xét: *sumdigit*(1234) = *sumdigit*(123) + 4, tổng các chữ số của *n* bằng tổng các chữ số của *n* sau khi bỏ đi hàng đơn vị cộng với giá trị của chữ số hàng đơn vị. Hơn nữa số có 1 chữ số thì có tổng các chữ số chính là số đó. Từ đó ta có cách tính đệ qui theo số lượng chữ số của tham số *n* như sau:

- *sumdigit*(*n*) = *n* nếu *n* < 10 (số nhỏ hơn 10 thì chỉ có 1 chữ số hàng đơn vị).
- *sumdigit*(*n*) = *sumdigit*(*n*/10) + (*n*%10) nếu *n* ≥ 10 (*n* % 10 được giá trị của chữ số hàng đơn vị và *n* / 10 được số bỏ đi chữ số hàng đơn vị).

Trong C ta có thể viết hàm `sumdigit` đệ qui như sau.

Mã 4.6.4 – Hàm đệ qui tính tổng các chữ số

```
int sumdigit(int n)
{
    if(n < 10)
```

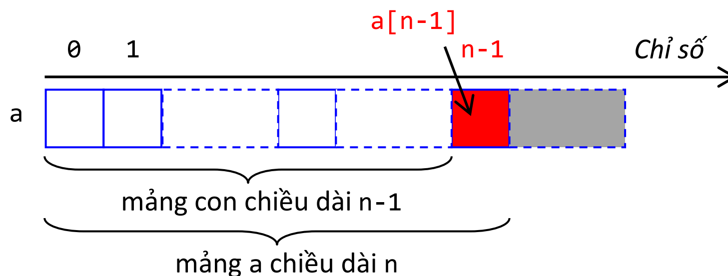
```
            return n;

        return sumdigit(n/10) + n%10;
    }
```

Liên quan đến số lượng thì một trường hợp đệ qui phổ biến khác theo số lượng đó là *đệ qui theo số lượng* các phần tử trong một mảng. Chẳng hạn giả sử ta muốn tính tổng các phần tử của một mảng *a* có *n* phần tử ta có thể tính theo cách đệ qui như sau:

- Nếu mảng có 1 phần tử thì tổng chính là phần tử duy nhất đó.
- Nếu mảng có *n* phần tử (*n* > 1) thì tổng chính là tổng các phần tử trong *n* – 1 phần tử đầu cộng với phần tử cuối cùng (phần tử ở vị trí *n* – 1). Lưu ý: *n* – 1 phần tử đầu của một mảng chính là mảng có *n* – 1 phần tử (ít hơn 1 phần tử, phần tử cuối, so với mảng ban đầu) mà ta gọi là "*mảng con*" như hình minh họa sau.

Hình 4.6.1 – Hình minh họa mảng con của mảng a chiều dài n



Từ đó ta có thể viết hàm đệ qui như sau.

Mã 4.6.5 – Hàm đệ qui tính tổng các phần tử trong mảng

```
int sum(int a[], int n)
{
    if(n == 1)
        return a[0];

    return sum(a, n - 1) + a[n - 1];
}
```

Ở trên ta đã không cho phép trường hợp mảng rỗng (*n* = 0). Thật ra trường hợp này cũng hay xuất hiện (như tập rỗng). Ta sửa lại trường hợp cơ sở của đệ qui để cho phép trường hợp mảng rỗng (với tổng là 0):

- Nếu mảng không có phần tử nào (*n* = 0) thì tổng là 0.

Từ đó viết lại hàm `sum` như sau.

Mã 4.6.6 – Hàm `sum` viết lại (cho phép mảng rỗng)

```
int sum(int a[], int n)
{
```

```
    if(n == 0)
        return 0;

    return sum(a, n - 1) + a[n - 1];
}
```

Còn những dạng khác của tham số đệ qui kiểu số nguyên. Bạn hãy xem lại cách vẽ thảm Sierpinski trong phần Mở rộng 4.2. Đó là một cách làm đệ qui. Ta đã *đệ qui theo độ rộng* (tham số `w` của hàm đệ qui `Sierpinski`, Mã 4.2.6) của thảm. Trường hợp cơ sở, độ rộng là 0, khi đó ta không cần làm gì cả (tưởng tưởng tấm thảm độ rộng 0), tuy nhiên đây là trường hợp quan trọng mà ta không được phép bỏ đi. Thực ra thì dạng đệ qui trong hàm `Sierpinski` là dạng đệ qui phức tạp hơn các dạng đệ qui trên do ta đã đệ qui nhiều lần (8 lần, để vẽ tấm thảm Sierpinski lớn ta đục hình vuông con ở giữa và vẽ tấm thảm Sierpinski với kích thước 1/3 vào 8 hình vuông con xung quanh). Điều đó có thực sự khác biệt? Ta có thể không dùng đệ qui để làm không? Nếu có thì đệ qui thực sự mang lại điều gì? … Những câu hỏi như vậy sẽ sáng tỏ khi bạn luyện lên tầng cao hơn.

MỞ RỘNG 4.6 – Bài toán Tháp Hà Nội

Bạn có biết số phận của ta (thậm chí là cả vũ trụ) nằm trong tay một nhà sư không? Tương truyền rằng trong một ngôi tháp nào đó ở Hà Nội (Tower of Hanoi[5]) có một nhà sư đang ngày đêm chuyển 64 cái đĩa vàng mà khi hoàn thành là lúc tận thế. Cụ thể là có *n* cái đĩa kích thước tăng dần và có 3 chồng đĩa mà ban đầu cả *n* cái đĩa đều đặt trên chồng 1 với đĩa lớn ở dưới, nhỏ ở trên. Nhà sư phải tìm cách chuyển toàn bộ *n* cái đĩa ở chồng 1 sang chồng 2 (có thể dùng chồng 3 làm trung gian) theo qui định:

- Mỗi lần chỉ được di chuyển một đĩa từ đỉnh một chồng này sang đỉnh một chồng khác.
- Không được để đĩa lớn ở trên đĩa nhỏ hơn.

Bạn tra cứu thêm về bài toán này nhé[6]. Bài toán này có thể được giải theo nhiều cách nhưng cách giải đệ qui là hay nhất. Để chuyển *n* đĩa từ chồng nguồn 1 sang chồng đích 2 (lấy chồng 3 làm trung gian) ta làm đệ qui như sau:

- Nếu $n = 0$ (không có đĩa nào): không làm gì cả.
- Nếu $n > 0$:
    - Chuyển $n - 1$ đĩa (ở trên) từ chồng 1 sang chồng 3 (lấy chồng

[5] Chắc không phải là tòa nhà Hà Nội Tower ở số 49 Hai Bà Trưng, Hoàn Kiếm nhỉ:)

[6] Có thể xem bài "Tower of Hanoi" trên trang Wikipedia (https://en.wikipedia.org/wiki/Tower_of_Hanoi).

2 làm trung gian).

- Chuyển 1 đĩa (dưới cùng) từ chồng 1 sang chồng 2.
- Chuyển $n - 1$ đĩa (trước đó) từ chồng 3 sang chồng 2 (lấy chồng 1 làm trung gian)

Sau đây là chương trình giúp nhà sư siêu độ chúng sinh nhanh hơn:)

Mã 4.6.7 – Chương trình giải bài toán Tháp Hà Nội bằng đệ qui

```
#include <stdio.h>

#define MAX 100
struct STACK {
    char name;
    int a[MAX];
    int num;
};

void init_stack(STACK &s, char name) {
    s.num = 0; s.name = name;
}
void push_stack(STACK &s, int e) {
    s.a[s.num++] = e;
}
int pop_stack(STACK &s) {
    return s.a[--s.num];
}
void print_stack(STACK &s) {
    for(int i = 0; i < s.num; i++)
        printf("%d ", s.a[i]);
}

STACK A, B, C;
int num_moves = 0;

void print_disks() {
    printf("\nStack %c: ", A.name); print_stack(A);
    printf("\nStack %c: ", B.name); print_stack(B);
    printf("\nStack %c: ", C.name); print_stack(C);
}

void move(int n, STACK &s1, STACK &s2, STACK &s3) {
    if(n == 0)
        return;

    move(n - 1, s1, s3, s2);

    push_stack(s2, pop_stack(s1));
```

```
40      num_moves++;
41      printf("\n\n%c -> %c", s1.name, s2.name);
42      print_disks();
43
44      move(n - 1, s3, s2, s1);
45  }
46
47  int main() {
48      int n;
49      printf("Number of disks? ");
50      scanf("%d", &n);
51
52      init_stack(A, 'A'); init_stack(B, 'B');
    init_stack(C, 'C');
53      for(int disk = n; disk >= 1; disk--)
54          push_stack(A, disk);
55      print_disks();
56      move(n, A, B, C);
57      printf("\n\nFinished in %d moves.\n", num_moves);
58
59      return 0;
60  }
```

Đầu tiên cấu trúc STACK khai báo ở Dòng 4 đến Dòng 8 giúp ta mô tả một chồng đĩa với các thành phần: name chứa tên của chồng (chẳng hạn chồng 1, 2, … hay chồng *A*, *B*, …), mảng a chứa các đĩa trong chồng (giả sử mỗi đĩa là một số nguyên, tức là đĩa 1, 2, …) và num cho biết số lượng đĩa trong chồng. Hàm init_stack (Dòng 10) khởi động chồng đĩa s với tên name được cho. Lưu ý s là tham số vào vì dữ liệu của s sẽ bị hàm thay đổi (đặt lại s.num và s.name). Hàm push_stack (Dòng 13) thêm đĩa e vào chồng s. Lưu ý đĩa phải được thêm vào đỉnh của chồng. Hàm pop_stack (Dòng 16) lấy đĩa khỏi chồng s. Lưu ý phải lấy đĩa từ đỉnh của chồng. Hàm print_stack (Dòng 19) giúp xuất ra các đĩa trong chồng s (theo thứ tự từ đáy lên đỉnh chồng). Lưu ý là hàm này không thay đổi dữ liệu của tham số s nhưng ta vẫn để s là tham chiếu (có dấu & trước tên tham số) để việc truyền nhận tham số dễ dàng. Nhớ lại rằng nếu để s làm tham trị (tham số bình thường, không có dấu & trước tên tham số) thì khi gọi hàm, toàn bộ dữ liệu của đối số (kiểu STACK) sẽ được chép vào s, nên rất tốn kém (bộ nhớ và thời gian) do kiểu STACK có một thành phần là mảng chứa nhiều số nguyên (thành phần a). Ngược lại, nếu dùng tham chiếu thì sẽ nhẹ nhàng hơn. Ta sẽ tìm hiểu kĩ vấn đề này sau còn bây giờ ta nhớ học hỏi cách làm này cho các cấu trúc có kích thước lớn (chẳng hạn có thành phần là mảng). Dĩ nhiên là ta phải tự mình đảm bảo rằng không được thay đổi dữ liệu của tham số.

Hàm quan trọng là hàm `move` (Dòng 33). Hàm này thực hiện công việc chuyển đĩa bằng cách đệ qui như đã mô tả ở trên. Dòng 34 ứng với trường hợp cơ sở mà ở đây đơn giản là kết thúc hàm bằng lệnh `return` ở Dòng 35 (nghĩa là không làm gì cả). Các dòng sau đó ứng với trường hợp đệ qui (n > 0) mà đầu tiên là Dòng 37 chuyển $n - 1$ đĩa từ chồng 1 sang chồng 3 (lấy chồng 2 làm trung gian) bằng lời gọi đệ qui. Dòng 39 chuyển 1 đĩa từ chồng 1 sang chồng 2. Dòng này viết khá cô đọng: `pop_stack(s1)` lấy đĩa (trên đỉnh) của chồng 1 và đĩa đó được đặt vào (đỉnh) chồng 2 bằng `push_stack`. Dòng 40 tăng biến đếm `num_moves` là biến đếm số lượng lần di chuyển đĩa. Đây là biến toàn cục, được khai báo và khởi động giá trị 0 ở Dòng 25. Ta để biến này là biến toàn cục vì nó được chia sẻ bởi nhiều hàm (hàm `move` và hàm `main`). Dòng 41 xuất ra thông báo chuyển (một) đĩa từ chồng 1 sang chồng 2 và Dòng 42 xuất ra tình trạng 3 chồng đĩa sau khi chuyển đĩa. Cuối cùng, lời gọi đệ qui ở Dòng 44 chuyển $n - 1$ đĩa từ chồng 3 sang chồng 2 (lấy chồng 1 làm trung gian).

Công việc của hàm `main` khá rõ ràng: cho người dùng chọn số đĩa *n*; khởi động 3 chồng đĩa *A*, *B*, *C* với chồng *A* chứa *n* đĩa lần lượt từ đáy lên đỉnh là đĩa n, $n - 1$, …, 1 còn chồng *B*, *C* không chứa đĩa nào; xuất ra tình trạng 3 chồng đĩa ban đầu; gọi hàm đệ qui `move` để chuyển *n* đĩa từ chồng *A* sang chồng *B* (lấy chồng *C* làm trung gian) và cuối cùng là in ra tổng số lần di chuyển (một) đĩa. Lưu ý là các biến chồng đĩa `A`, `B`, `C` cũng được khai báo toàn cục (ở Dòng 24) vì ngoài hàm `main` thì hàm `print_disks` (Dòng 27) cũng dùng chúng.

Cách làm đệ qui trên không chỉ đơn giản, dễ hiểu mà còn rất hiệu quả. Thật vậy, nó tốn $2^n - 1$ lần chuyển (một) đĩa và là số lần tốt nhất (ít nhất có thể) mà người ta đã chứng minh được. Bạn chạy thử sẽ thấy nó tốn 3 lần ($2^2 - 1$) cho 2 đĩa, 7 lần ($2^3 - 1$) cho 3 đĩa, 15 lần ($2^4 - 1$) cho 4 đĩa, … Nhưng nhớ là đừng chạy thử với 64 đĩa nhé, nếu không khi chương trình kết thúc cũng là lúc bạn tiễn mọi người lên đường đó[7]:) Thật ra thì người ta đã tính được nếu tốn 1 giây để chuyển (một) đĩa thì nhà sư cần khoảng 585 tỉ năm để hoàn thành (chuyển 64 đĩa). Hú hồn!:)

BÀI TẬP

Bt 4.6.1 Người ta thường qui ước 0! = 1[8]. Hãy sửa hàm đệ qui tính giai thừa ở bài học trên (Mã 4.6.1) để cho phép tính giai thừa của số nguyên không âm. (*Gợi ý*: xem lại trường hợp cơ sở.)

Bt 4.6.2 Viết lại Mã 1.5.5 bằng kĩ thuật đệ qui.

---

[7] Nói vậy chứ nhớ phím tắt Ctrl + C (nhấn giữ đồng thời Ctr và C) để ngắt chương trình Console đang chạy.

[8] Thật ra không thể qui ước khác được:)

Bt 4.6.3 Viết lại Mã 1.5.6 bằng kĩ thuật đệ qui.

Bt 4.6.4 Làm lại Bài tập 1.5.6 bằng kĩ thuật đệ qui.

Bt 4.6.5 Làm lại Bài tập 2.5.2 bằng kĩ thuật đệ qui.

Bt 4.6.6 Làm lại Bài tập 2.5.5 bằng kĩ thuật đệ qui.

Bt 4.6.7 Làm lại Bài tập 3.2.3(d), (f), (g) bằng kĩ thuật đệ qui. So sánh tính hiệu quả (thời gian thực thi) của kĩ thuật đệ qui với kĩ thuật lặp.

Bt 4.6.8 Trong Bài tập 3.3.3 ta đã tính số tổ hợp bằng kĩ thuật lặp. Hãy tra cứu để biết cách định nghĩa đệ qui số tổ hợp (còn gọi là hệ số nhị thức, binomial coefficient) và viết hàm đệ qui cài đặt định nghĩa này. So sánh tính hiệu quả với cài đặt bằng kĩ thuật lặp ở Bài tập 3.3.3.

Bt 4.6.9 Tất cả các thao tác trên mảng đều có thể cài đặt bằng kĩ thuật đệ qui. Hãy làm lại Bài tập 4.1.7 bằng kĩ thuật đệ qui.

Bt 4.6.10 Trong Bài tập 4.1.8 ta đã biết vài thuật toán sắp xếp đơn giản. Hãy cài đặt thuật toán sắp xếp chèn (Insertion Sort) bằng kĩ thuật đệ qui. Một thuật toán sắp xếp rất hiệu quả có bản chất đệ qui là *Quicksort*. Hãy tra cứu để nắm được thuật toán này và cài đặt nó.

Bt 4.6.11 Chuỗi là mảng đặc biệt (mảng kí tự kết thúc bởi kí tự NULL). Bạn đã làm đệ qui được trên mảng. Vậy bạn có làm đệ qui được trên chuỗi không? Thử làm lại Bài tập 4.4.1 bằng đệ qui. Nếu được, bạn thử làm tiếp Bài tập 4.4.4 bằng đệ qui. (Còn nếu không thì đừng lo lắng, bạn sẽ làm được dễ dàng khi luyện qua tầng sau.)

Bt 4.6.12 Viết hàm đệ qui tìm phân số lớn nhất trong một mảng phân số để thấy rằng nó cũng đơn giản như là đệ qui trên mảng số nguyên. (Phân số được mô tả bởi kiểu cấu trúc trong Bài tập 4.5.1.)

Đệ qui là công cụ vô giá của C!

BÀI 4.7

Ở Bài 4.1 ta đã viết chương trình tìm số lớn nhất trong trong dãy *n* số nguyên mà người dùng nhập vào. Trường hợp *n* nhỏ (số lượng ít) thì người dùng có thể chịu khó ngồi nhập tay từ bàn phím được nhưng trường hợp *n* lớn (chẳng hạn vài ngàn hay vài triệu) thì rõ ràng không người dùng nào có thể ngồi gõ tay chừng ấy con số được. Hơn nữa trong thời đại tràn ngập dữ liệu như ngày nay thì dữ liệu có thể đến từ nhiều nguồn khác nhau (từ các chương trình khác, từ trên mạng, từ người khác, từ máy khác, …). Trong những trường hợp như vậy ta có thể để dữ liệu trong *tập tin* và chương trình của ta, thay vì lấy dữ liệu từ người dùng, sẽ lấy dữ liệu từ tập tin và xử lý. Dữ liệu được lưu trữ trên tập tin có đặc điểm quan trọng là *thường trú* trên đó, nghĩa là nó vẫn còn khi máy tắt, hơn nữa nó có thể được sao chép từ máy này sang máy khác và có thể chứa lượng rất lớn dữ liệu (có thể chứa hàng tỉ tỉ số, vượt xa con số mà người dùng có thể nhập liệu từ bàn phím).

Trong bài này ta sẽ làm việc với dạng thức tập tin đơn giản nhất là *tập tin văn bản*. Chính xác hơn gọi là *tập tin thuần văn bản*: dữ liệu trong tập tin chính là dãy các kí tự, tức là chuỗi. Lưu ý rằng thuật ngữ văn bản ở đây không phải là văn bản như Word. Trong Word thì văn bản được định dạng với các thông tin như kiểu dáng (đậm, nghiêng, …), font chữ, màu sắc, … văn bản của ta ở đây chỉ là dãy kí tự mà thôi. Có nhiều chương trình cho phép ta tạo và đọc tập tin văn bản. Chẳng hạn Notepad trên Windows. Bạn hãy tạo một tập tin văn bản tên `number.txt` để ở ổ đĩa `D` (tức là có tập tin văn bản với đường dẫn `D:\number.txt`) với nội dung sau[1]:

```
10
1 -2 30 44 -55 666 -77 888 9999 1010
```

Con số đầu tiên cho biết số lượng số (*n*) và *n* con số sau đó là các số nguyên. Ở đây *n* là 10 và 10 con số sau đó chỉ để ví dụ, bạn nhập số (nguyên) nào cũng được nhé. Hơn nữa con số đầu tiên (*n*) đã được để riêng trên dòng thứ nhất và *n* con số còn lại để trên dòng thứ hai với các số cách nhau bằng khoảng trắng. Bạn có thể để mỗi số trên một dòng hoặc tất cả trên một dòng, … đều được miễn sao giữa các số có khoảng trắng (space, tab, xuống dòng đều là khoảng trắng).

Chương trình sau đây tìm số lớn nhất trong *n* số nguyên đọc từ tập tin văn bản trên. Bạn hãy gõ và chạy thử.

---

[1] Xem Phụ lục A.8 để biết cách tạo tập tin văn bản.

Mã 4.7.1 – Chương trình tìm số lớn nhất trong *n* số nguyên đọc từ tập tin

```
1  #include <stdio.h>
2  #include <limits.h>
3  int main()
4  {
5      FILE* file = fopen("D:\\number.txt", "r");
6      if(file == 0)
7          printf("Khong the mo duoc file");
8      else
9      {
10         int n;
11         fscanf(file, "%d", &n);
12
13         int x, M = INT_MIN;
14         for(int i = 1; i <= n; i++)
15         {
16             fscanf(file, "%d", &x);
17             if(x > M)
18                 M = x;
19         }
20         printf("So lon nhat la: %d\n", M);
21
22         fclose(file);
23     }
24
25     return 0;
26 }
```

Để có thể lấy (đọc) dữ liệu trong tập tin thì bước đầu tiên là *mở tập tin*. Ta dùng hàm `fopen` (trong `stdio.h`) để làm điều này như Dòng 5 cho thấy. Tham số đầu tiên là *đường dẫn* của tập tin, là chuỗi định vị tập tin trên đĩa, ở đây là đường dẫn `D:\number.txt`. Lưu ý là để mô tả kí tự \ trong hằng chuỗi của C thì ta dùng \\. Tham số thứ 2 xác định *chế độ mở tập tin*, tức là cách mở tập tin, ở đây là `"r"` xác định chế độ mở tập tin để đọc văn bản (`r` là read as text)[2]. Hàm `fopen` trả về cái gọi là *con trỏ file*. Bạn có thể hình dung đó là cái giúp ta quản lý file (chẳng hạn ta sẽ dùng nó để đọc, ghi, đóng, … tập tin). Về con trỏ ta sẽ tìm hiểu ở tầng cao hơn nhưng con trỏ file có kiểu là `FILE*` nên ở trên ta khai báo biến `file` kiểu là `FILE*` để chứa giá trị trả về của `fopen` tức là chứa con trỏ file của file `number.txt`. Hiển nhiên là bạn có thể dùng tên khác cho biến `file`.

Bước kế tiếp là ta kiểm tra có mở được file không. Nếu không mở được file (chẳng hạn do đường dẫn không đúng hoặc file đó bị hư hoặc bị cấm truy cập, …) thì `fopen` sẽ trả về 0 còn nếu mở được thì nó mới trả về con trỏ file. Do đó lệnh `if` ở Dòng 6 kiểm tra điều này, trong thân `else` (nghĩa là giá

[2] Có các chế độ mở tập tin khác mà bạn tự tra cứu để biết nhé.

trị trả về của fopen khác 0) thì ta mở được file và biến file chứa con trỏ file, khi đó ta dùng con trỏ file chứa trong biến file để xử lý tập tin number.txt.

Bước kế tiếp là xử lý tập tin, ở đây là *đọc tập tin*. Ta cần đọc số đầu tiên trong tập tin vào biến n. Dòng 11 làm điều này bằng cách dùng hàm fscanf. Hàm fscanf có cách dùng hoàn toàn giống scanf trừ việc nó nhận thêm đối số đầu tiên là con trỏ file. Hàm fscanf sẽ nhập từ file quản lý bởi con trỏ file còn scanf thì nhập từ người dùng (từ bàn phím). Ta gặp lại fscanf ở Dòng 16. Lệnh đó giúp ta đọc các số từ tập tin vào biến x. Như vậy fscanf cũng sẽ đọc lần lượt từ chuỗi trong tập tin giống như cách scanf đọc lần lượt từ chuỗi nhập của người dùng (buffer). Thật ra là hoàn toàn như nhau nếu bạn nghĩ rằng buffer của fscanf chính là chuỗi trong tập tin tương ứng.

Bước cuối cùng, sau khi xử lý xong tập tin, ta *đóng tập tin* lại. Hàm fclose nhận một tham số là con trỏ tập tin cần đóng và nó sẽ đóng tập tin đó lại. Dòng 22 cho thấy cách dùng fclose. Khi ta đóng tập tin thì nó sẽ được giải phóng, chương trình của ta sẽ không chiếm dụng tập tin đó nữa (nghĩa là chương trình khác có thể dùng tập tin đó) và cũng có nghĩa là ta không được xử lý trên tập tin đó nữa.

Mã trên là khuôn mẫu các bước để xử lý tập tin và ta đã dùng nó để đọc dữ liệu từ tập tin. Một thao tác xử lý khác trên tập tin là *ghi dữ liệu vào tập tin* (nghĩa là tạo tập tin chứa dữ liệu mong muốn). Chẳng hạn ta sẽ tạo một tập tin chứa một lượng lớn số nguyên phát sinh ngẫu nhiên và tập tin này có thể được dùng để kiểm tra chương trình tìm số lớn nhất trên.

Mã 4.7.2 – Chương trình tạo tập tin chứa các số nguyên ngẫu nhiên

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    char filename[100];
    printf("Nhap duong dan tap tin muon tao: ");
    scanf("%s", filename);

    FILE *file = fopen(filename, "w");
    if(file == 0)
        printf("Khong the tao duoc file");
    else
    {
        int n;
        printf("Nhap so luong so se tao ngau nhien: ");
        scanf("%d", &n);
        fprintf(file, "%d\n", n);

        srand(time(0));
```

```
21            for(int i = 1; i <= n; i++)
22            {
23                int x = rand();
24                fprintf(file, "%d ", x);
25            }
26
27            fclose(file);
28        }
29
30        return 0;
31    }
```

Chương trình trên cho người dùng nhập đường dẫn file muốn tạo vào chuỗi `filename`. Chẳng hạn nếu muốn tạo file `number.txt` trong ổ đĩa `D` thì người dùng sẽ gõ `D:\number.txt` (lưu ý là chỉ có một dấu \ thôi). Sau đó chương trình theo các bước như trên để làm việc với file. Nhớ là ở đây ta ghi dữ liệu ra file chứ không phải đọc dữ liệu từ file. Các khác biệt như vậy là:

- Trong tham số thứ 2 của `fopen` ta truyền chuỗi `"w"` để tạo tập tin văn bản (`w` là viết tắt của write text) như Dòng 10 cho thấy.
- Ta dùng hàm `fprintf` để ghi dữ liệu vào file. Dòng 18 ghi số nguyên trong `n` (và xuống dòng) vào tập tin quản lý bởi con trỏ file trong biến `file`. Tương tự như `printf` nhưng hàm `fprintf` có thêm tham số đầu tiên là con trỏ file mà ta muốn xuất dữ liệu ra đó còn `printf` thì xuất dữ liệu ra màn hình (cửa sổ Console). Ta gặp lại `fprintf` ở Dòng 24. Lệnh đó giúp ta ghi các số `x` (và khoảng trắng) vào tập tin. Như vậy `fprintf` sẽ xuất dữ liệu nối tiếp vào tập tin giống như cách `printf` xuất dữ liệu nối tiếp ra màn hình.

Mã trên hơi rắc rối ở chỗ: các số nguyên ta xuất ra file là phát sinh ngẫu nhiên. Như vậy ta sẽ tạo số ngẫu nhiên và bỏ vào `x`. Để tạo số nguyên ngẫu nhiên (nghĩa là số nguyên nào đó không đoán trước được) bạn dùng hàm `rand` không tham số như Dòng 23. Hàm này có trong `stdlib.h`. Trước khi dùng hàm này bạn cần khởi tạo bộ tạo số ngẫu nhiên bằng hàm `srand` cũng trong `stdlib.h`. Hàm này cần một tham số là giá trị nguyên nào đó gọi là hạt giống của bộ tạo số. Để hạt giống này khác nhau trong mỗi lần chạy (nghĩa là sẽ được các số ngẫu nhiên khác nhau trong mỗi lần chạy khác nhau của chương trình) ta dùng hàm `time` trong `time.h`. Hàm `time` với đối số 0 sẽ trả về thời gian hiện hành (và do đó khác nhau mỗi lần chương trình chạy). Hầy… Tóm lại gõ y chang Dòng 20 là được (và nhớ `#include stdlib.h` và `time.h`). Ở đây không quan trọng chuyện này. Quan trọng là đọc/ghi file văn bản nhé. Bạn hãy chạy chương trình trên (Mã 4.7.2) để tạo file văn bản chứa các số ngẫu nhiên và dùng chương trình trước đó (Mã 4.7.1) để tìm số lớn nhất trong file đã tạo.

Bây chừ ta sẽ viết chương trình dùng cả hai thao tác đọc và ghi file. Chương trình này sẽ đọc dữ liệu là dãy các số nguyên từ file văn bản mà ta gọi là *file input*. Sau đó chương trình sẽ tạo ra một file văn bản mà ta gọi là *file output* chứa các số nguyên trong file input nhưng theo thứ tự giảm dần. Khuôn mẫu của chương trình như sau.

Mã 4.7.3 – Khuôn mẫu chương trình xử lý dữ liệu trên bộ nhớ ngoài

```
#include <stdio.h>
int main()
{
    int n, a[10000];
    docfile("D:\\number.txt", a, n);
    sapxep(a, n);
    ghifile("D:\\number2.txt", a, n);

    return 0;
}
```

Điều quan trọng trong khuôn mẫu trên là ta cần nhập (đọc) dữ liệu từ file input vào thành dữ liệu của chương trình, ở đây là mảng số nguyên, sau đó xử lý trên dữ liệu của chương trình và xuất (ghi) dữ liệu vào file output. Trong khi file được lưu trữ trên *bộ nhớ ngoài* (như đĩa cứng, đĩa USB, …) thì dữ liệu của chương trình (như các biến hay mảng, …) thì được để trên các vùng nhớ trên *bộ nhớ trong* (còn gọi là bộ nhớ của chương trình). Ta không thể trực tiếp thao tác trên dữ liệu ở bộ nhớ ngoài. Ta sẽ trở lại với việc xử lý file ở tầng cao hơn. Hiện giờ thì chương trình trên là khuôn mẫu điển hình:

- Ta cần vùng nhớ trong chương trình để chứa dữ liệu lấy từ file, ở đây là mảng các số nguyên a (với sức chứa đủ lớn) để chứa các số nguyên lấy từ file input. Chương trình cũng có các vùng nhớ chứa dữ liệu của riêng của nó như thông thường. Ở trên biến `n` sẽ giúp quản lý số lượng số nguyên lấy từ file input (và cũng sẽ là chiều dài mảng).
- Đọc dữ liệu từ file input vào chương trình. Ở trên hàm `docfile` sẽ đọc các số nguyên trong file input (cụ thể là file `D:\number.txt`) vào mảng a (số lượng không được quá sức chứa của mảng) và đặt giá trị của n là số lượng số đọc vào, chính là chiều dài của mảng a.
- Xử lý dữ liệu của chương trình. Ở trên thì hàm `sapxep` sẽ giúp sắp xếp mảng a theo thứ tự mong muốn. Hàm này có prototype:

  ```
  void sapxep(int a[], int n)
  ```

  Tùy theo cách sắp xếp mà bạn tự cài đặt hàm này nhé (xem lại Bài 4.1)

- Ghi dữ liệu từ chương trình vào file. Ở trên hàm `ghifile` sẽ lấy các số trong mảng a (với n là số lượng số) và ghi vào file output (cụ thể là tập tin `D:\number2.txt`).

Hiển nhiên là ta phải có file văn bản `D:\number.txt` trên đĩa, trong đó chứa các số nguyên và ta cũng có thể dùng đường dẫn file khác hoặc có thể cho người dùng nhập các đường dẫn file input và output mong muốn. Hơn nữa đường dẫn file output có thể trùng với file input. Khi đó file input (cũ) sẽ được ghi đè bởi file output (mới) mà hiệu ứng giống như là ta sắp xếp số nguyên trong chính file input. Thật ra mô hình nhập, xử lý và xuất dữ liệu ta đã thấy từ rất sớm, khác là ở đây ta nhập/xuất dữ liệu từ file thay vì từ người dùng. Mô hình tương tác của chương trình với môi trường xung quanh (người dùng/hệ điều hành/chương trình khác, …) rất đa dạng, từ đơn giản đến phức tạp mà ta sẽ tìm hiểu sau.

Có những cách nào giúp ta quản lý tất cả các phần tử của mảng (từ đó giúp xử lý tất cả các phần tử). Với những mảng thông thường, ta hay dùng kèm một thông tin là chiều dài mảng: xuất phát từ đầu mảng theo chiều dài này ta sẽ biết được vị trí kết thúc. Ngược lại, với chuỗi ta dùng kí tự `NULL` để đánh dấu kết thúc chuỗi: xuất phát từ đầu chuỗi cho đến khi ta đụng vị trí được đánh dấu. Đây cũng là hai hình thức phổ biến (mặc dù còn những cách khác phù hợp cho các tình huống khác nhau). Tuy nhiên xem ra cách thứ hai thì phổ biến hơn. Khi bắt đầu một cuộc trò chuyện (qua phone chẳng hạn), ta thường không nói trước rằng ta sẽ nói chuyện bao lâu mà ta thường nói đến khi hết chuyện thì thôi và có thể có kết thúc bằng kí hiệu đánh dấu "Bye" (hoặc thậm chí không có). Tình huống này là bắt buộc khi mà ta không thể xác định trước "số lượng" hay "chiều dài". Với file input chứa các số nguyên với cấu trúc như ở trên, tức là số đầu tiên cho biết số lượng số (*n*) và *n* số sau đó mới là các số thực sự cần xử lý thì ta có thể đọc dữ liệu tương tự Mã 4.7.1. Tuy nhiên tình huống phổ biến hơn là ta không có con số cho biết số lượng trong file, nghĩa là file input chỉ chứa đúng các số cần xử lý (mà không chứa thông tin số lượng số, không có *n*). Trong trường hợp như vậy ta có cách nhập như minh họa trong hàm `docfile` sau.

Mã 4.7.4 – Hàm đọc dữ liệu từ file khi không biết trước số lượng

```
int docfile(char input[], int a[], int &n)
{
    FILE* file = fopen(input, "r");
    if(file == 0)
        return 0;

    int x;
    n = 0;
    while(fscanf(file, "%d", &x) > 0)
    {
        a[n] = x;
```

```
12          n++;
13      }
14      fclose(file);
15
16      return 1;
17  }
```

Ta làm sao đọc hết các số trong file input khi mà ta không biết số lượng. Nếu biết số lượng thì ta sẽ đọc đúng chừng đó lần (bằng một vòng lặp `for` như Dòng 14 Mã 4.7.1). Dễ thôi, ta cứ đọc vào số nguyên cho đến khi không đọc được nữa (nghĩa là ta đã đọc hết file). Làm sao biết là ta đọc được hay không? Rất đơn giản, các hàm nhập từ file (và các hàm nhập khác) đều có cách này kia cho biết được điều này chẳng hạn hàm `fscanf` (và `scanf`) sẽ trả về số lượng dữ liệu chuyển đổi được (nếu chuyển đổi được hết thì con số trả về sẽ bằng số lượng đặc tả chuyển đổi (%...) trong chuỗi định dạng). Chẳng hạn lời gọi hàm `fscanf` ở Dòng 9 `fscanf(file, "%d", &x)` sẽ đọc tiếp một số nguyên từ file quản lý bởi con trỏ file trong biến `file`. Nếu đọc được sẽ trả về 1 còn không đọc được sẽ trả về 0. Như vậy ta có thể dùng con số trả về này để biết là đã đọc hết hay chưa. Nếu tinh ý thì vòng `while` ở trên có thể viết gọn hơn là:

```
while(fscanf(file, "%d", &x))
    a[n++] = x;
```

Cũng lưu ý là tham số n ở trên là tham số ra nên có dấu & trước tên tham số (Dòng 1). Hàm `docfile` trả về đúng (1) khi đọc được dữ liệu từ file input còn ngược lại thì trả về sai (0).

Hàm `ghifile` không ghi số lượng số (*n*) ra file mà chỉ ghi các số thôi. Hàm này trả về đúng (1) khi ta ghi được dữ liệu ra file output còn ngược lại thì trả về sai (0).

Mã 4.7.5 – Hàm ghi file không ghi trước số lượng

```
int ghifile(char output[], int a[], int n)
{
    FILE* file = fopen(output, "w");
    if(file == 0)
        return 0;

    for(int i = 0; i < n; i++)
        fprintf(file, "%d ", a[i]);
    fclose(file);

    return 1;
}
```

Ta cũng có thể dùng chương trình phát sinh các số ngẫu nhiên ở Mã 4.7.2 để tạo ra một file văn bản chứa lượng lớn các số ngẫu nhiên sau đó

dùng file này làm file input kiểm tra lại chương trình sắp xếp trên (Mã 4.7.3). Khi đó nhớ là không có xuất số lượng số (*n*) vào file, nghĩa là bỏ Dòng 18 ở Mã 4.7.2.

Bạn có thấy rằng hàm `fscanf` và `scanf` hay `fprintf` và `printf` rất giống nhau. Tương tự bạn có thể đoán rằng ta có hàm `fgets`, `fputs` mà hoạt động giống như `gets` và `puts`. Thật vậy, nếu như các hàm nhập xuất thông thường là nhập xuất từ người dùng (nhập từ bàn phím và xuất ra màn hình) thì các hàm nhập xuất từ file sẽ có thêm tham số là con trỏ đến file mà nó sẽ nhập/xuất từ/vào đó. Những hàm này có tên với tiếp đầu ngữ `f` đằng trước (`f` là viết tắt của file). Vậy thì nhập/xuất từ người dùng khác biệt gì với nhập xuất từ file. Nếu xuất vào mành hình thì dữ liệu (mà đều trở thành các kí tự) sẽ được đưa ra cửa sổ Console còn xuất vào file văn bản thì dữ liệu (mà đều trở thành các kí tự) sẽ được đưa ra file. Như vậy, một cách trừu tượng ta có thể xem màn hình (chứa kết quả xuất) là một file đặc biệt. Thật vậy C[3] xem nó như là một file với con trỏ file được `#define` trong hằng tượng trưng `stdout`. `stdout` có kiểu là `FILE*` được `#define` trong `stdio.h` và có thể được dùng trong toàn bộ chương trình khi ta đã `#include` thư viện này. Bạn có thể mường tượng là các hàm như `printf`, `puts`, … sẽ gọi các hàm `fprintf`, `fputs`, … với đối số thứ nhất là `stdout`. Như vậy ở đây ta có một kỹ thuật rất hữu ích trong một vài tình huống gọi là kĩ thuật *điều hướng*. Để điều hướng kết quả xuất thông thường (ra `stdout` mà bình thường là ra màn hình) ra file ta dùng hàm `freopen`. Chẳng hạn trong phần Mở rộng 4.2 ta đã vẽ thảm Sierpinski rất đẹp ra màn hình. Bây chừ ta sẽ "vẽ ra file" bằng kĩ thuật điều hướng như sau.

Mã 4.7.6 – Chương trình vẽ thảm Sierpinski ra file

```
1  int main()
2  {
3      freopen("D:\\output.txt", "w", stdout);
4  
5      char buff[MAX][MAX];
6      int n = 81;
7      fillblank('*', buff, n);
8      Sierpinski(' ', buff, 0, 0, n);
9      display(buff, n);
10 
11     fclose(stdout);
12 
13     return 0;
14 }
```

Lời gọi hàm `freopen` ở Dòng 3 giúp ta đặt lại `stdout` là con trỏ file đến file `D:\output.txt`. Mọi kết xuất sau đó đến `stdout` (chẳng hạn như kết

[3] Thật ra là hệ điều hành.

xuất từ `printf`, ...) đều sẽ vào file tương ứng. Khi không muốn kết xuất ra file nữa thì ta dùng `fclose` để đóng file lại như Dòng 11.

Tương tự ta có các hàm nhập thông thường sẽ lấy dữ liệu từ bàn phím và được quản lý bởi `stdin` và ta cũng có thể điều hướng nhập từ file bằng `freopen`. Chẳng hạn từ chương trình tìm số lớn nhất trong *n* số mà người dùng nhập (Mã 4.1.1), ta có thể viết chương trình tìm số lớn nhất trong *n* số đọc từ tập tin văn bản `D:\number.txt` bằng cách điều hướng nhập từ file rất đơn giản mà không cần phải làm như Mã 4.7.1. Thật vậy chỉ cần thêm lệnh `freopen("D:\\number.txt", "r", stdin);` và lệnh `fclose(stdin);` vào Mã 4.1.1 ta sẽ được chương trình tương đương Mã 4.7.1. Hiển nhiên là ta phải thêm vào chỗ phù hợp. Hi vọng là bạn thấy được chỗ điền 2 lệnh trên.

Lệnh đầu tiên giúp ta đặt lại `stdin` là con trỏ file đến file `D:\number.txt`. Mọi lời gọi nhập sau đó từ `stdin` (chẳng hạn như `scanf`, ...) đều sẽ lấy dữ liệu từ file tương ứng. Hiển nhiên là các thông báo nhập cho người dùng sẽ không cần thiết nữa và file nhập phải có cấu trúc như mong đợi: ở đây là số nguyên đầu tiên cho biết số lượng (*n*) và *n* số nguyên sau đó là là *n* số nguyên mà ta muốn tìm số lớn nhất trong đó. Các số nguyên cách nhau bằng khoảng trắng (có thể nhiều kí tự space, tab hay xuống dòng). Hơn nữa khi `stdin` là bàn phím thì nếu khi ta nhập hết dữ liệu trong chuỗi nhập thì chương trình sẽ bị treo để đợi người dùng nhập còn khi nhập từ file thì chương trình không bị treo, đơn giản là các hàm nhập sẽ trở về mà không đọc được dữ liệu, ta có thể dùng kết quả trả về để biết có đọc được không như ta đã làm trước đó.

MỞ RỘNG 4.7 – Tập tin CSV

Tập tin văn bản là dạng tập tin đơn giản nhất nhưng cũng phức tạp nhất. Về mặt vật lý nó chỉ là dãy các kí tự nhưng về logic nó có thể giúp lưu trữ mọi dạng dữ liệu tùy theo cách các kí tự được tổ chức. Ta sẽ tìm hiểu những dạng tập tin văn bản phức tạp như vậy sau (chẳng hạn HTML, XML, ...). Ở đây ta sẽ tìm hiểu một dạng tập tin văn bản khá đơn giản và được dùng phổ biến cho việc lưu trữ dữ liệu là *bảng tính*. Chẳng hạn ta có dữ liệu điểm thi của các sinh viên được cho trong Bảng 4.7.1. Bảng này chứa dữ liệu của 10 sinh viên với dữ liệu của mỗi sinh viên gồm 3 thành phần là mã số sinh viên (MSSV), điểm thi giữa kì (Diem GK) và điểm thi cuối kì (Diem CK). Các thành phần dữ liệu là các *cột* của bảng (còn gọi là các *field*). Dòng đầu tiên của bảng là *dòng tiêu đề*, cho biết tên của các field. Các dòng sau đó thì mỗi dòng cho biết dữ liệu của một sinh viên (còn gọi là các *record*).

Bảng tính là dạng tổ chức và lưu trữ dữ liệu rất phổ biến. Nó cũng được rất nhiều phần mềm hỗ trợ như Microsoft Excel, ... Mỗi phần mềm

thường có cách lưu trữ riêng cho bảng tính trong các file có định dạng riêng khác nhau. Tuy nhiên hầu hết đều hỗ trợ lưu trữ bảng tính ở dạng file văn bản khá đơn giản là *CSV*. Như tên gọi, Comma-Separated Values, đây là dạng file văn bản dùng dấu phẩy (,) để phân cách các trường dữ liệu[4]. Chẳng hạn bảng tính trên được lưu trữ trong file CSV sẽ có nội dung như Hình 4.7.1[5].

Bảng 4.7.1 – Bảng số liệu điểm thi của sinh viên

MSSV	Diem GK	Diem CK
1511001	4	4
1511002	5.5	6
1511003	10	10
1511004	9	8
1511005	4.5	5
1511006	3	10
1511007	4.5	5
1511008	8	8
1511009	9.5	2
1511010	9	8

Hình 4.7.1 – Bảng số liệu điểm thi được lưu trữ ở định dạng CSV

```
MSSV,Diem GK,Diem CK
1511001,4,4
1511002,5.5,6
1511003,10,10
1511004,9,8
1511005,4.5,5
1511006,3,10
1511007,4.5,5
1511008,8,8
1511009,9.5,2
1511010,9,8
```

Giả sử văn bản định dạng CSV trên được lưu trữ trong file `diem.csv` để trên ổ đĩa D (tức là file có đường dẫn `D:\diem.csv`) thì chương trình minh họa sau sẽ đọc dữ liệu điểm của các sinh viên từ file và hiển thị bảng điểm đó.

Mã 4.7.7 – Chương trình minh họa đọc dữ liệu từ file CSV

```c
#include <stdio.h>
```

[4] Tham khảo "Comma-separated values" trên trang Wikipedia (https://en.wikipedia.org/wiki/Comma-separated_values).
[5] Xem Phụ lục A.9 để biết cách tạo tạo file CSV với Microsoft Excel.

```

struct SinhVien {
    int mssv;
    float diemGK;
    float diemCK;
};

int docfilediem(char input[], SinhVien a[], int &n) {
    FILE* file = fopen(input, "r");
    if(file == 0)
        return 0;

    char line[100];
    fgets(line, 100, file);
    n = 0;
    while(fgets(line, 100, file)) {
        if(sscanf(line, "%d , %f , %f",
          &a[n].mssv, &a[n].diemGK, &a[n].diemCK) < 3)
            break;

        n++;
    }
    fclose(file);

    return 1;
}

void xuatdssv(SinhVien a[], int &n) {
    printf("|    MSSV | Diem GK | Diem CK | Diem TK |\n");
    printf("|--------------------------------------|\n");
    for(int i = 0; i < n; i++) {
        float diemTK = 0.3*a[i].diemGK + 0.7*a[i].diemCK;
        printf("|%8d | %7.1f | %7.1f | %7.1f |\n",
         a[i].mssv, a[i].diemGK, a[i].diemCK, diemTK);
    }
}

int main() {
    SinhVien a[200]; int n;
    if(!docfilediem("D:\\diem.csv", a, n))
        printf("Khong doc duoc file du lieu diem");
    else
        xuatdssv(a, n);

```

```
46        return 0;
47    }
```

Đầu tiên, Dòng 3 đến Dòng 7 khai báo kiểu cấu trúc `SinhVien` với các thành phần tương ứng chứa dữ liệu của một sinh viên. Để đơn giản ta đã dùng kiểu số nguyên cho mã số sinh viên (`mssv`)[6]. Hơn nữa vì điểm thi chỉ lẻ một chữ số nên ta dùng kiểu `float` để tiết kiệm cho `diemGK` và `diemCK`. Trong hàm `main`, ở Dòng 40, ta khai báo mảng `SinhVien a` để chứa dữ liệu của các sinh viên (tối đa 200 sinh viên) và biến nguyên `n` chứa số lượng sinh viên. Sau đó ta dùng hàm `docfilediem` để đọc dữ liệu điểm của các sinh viên trong file CSV `D:\diem.csv` vào mảng các sinh viên `a` (và `n` chứa số sinh viên đọc được). Nếu việc đọc dữ liệu thành công thì ta sẽ xuất ra danh sách các sinh viên với dữ liệu tương ứng bằng hàm `xuatdssv`.

Trong hàm `docfilediem` sau khi mở file ta dùng hàm `fgets` để đọc qua từng dòng dữ liệu trong file CSV. Sau khi bỏ qua dòng tiêu đề (dòng chứa tên các field) bằng Dòng 15 thì vòng lặp `while` giúp ta đọc qua từng dòng dữ liệu của mỗi sinh viên. Với từng dòng dữ liệu (dòng văn bản), ta dùng `fgets` để đọc vào chuỗi `line` (Dòng 17) rồi dùng `sscanf` để đọc mã số sinh viên, điểm thi giữa kì và điểm thi cuối kì. Hàm `sscanf` tương tự như `scanf` hay `fscanf` nhưng đọc từ một chuỗi thay vì từ bàn phím hay file. Bạn hãy tra cứu hàm này (cùng với đối tác của nó là hàm `sprintf`) để hiểu rõ hơn nhé. Bạn cũng coi lại Bài 3.1 nếu chưa thành thạo việc nhập có định dạng. Cũng giống như `scanf` (hay `fscanf`), `sscanf` sẽ trả về số mục chuyển đổi thành công, do đó nếu kết quả trả về nhỏ hơn 3 thì nghĩa là file dữ liệu không đúng định dạng. Cũng nhớ là `fgets` sẽ trả về 0 nếu không đọc thêm được dòng văn bản nào, mà điều đó có nghĩa là ta đã đọc hết file.

Hàm `xuatdssv` hiển thị danh sách sinh viên với 3 cột là mã số sinh viên, điểm thi giữa kì và điểm thi cuối kì. Ngoài ra, cột điểm tổng kết (`Diem TK`) cho biết điểm tổng kết của sinh viên. Dữ liệu này không có trong file dữ liệu sinh viên mà là dữ liệu tính được từ các dữ liệu khác. Cụ thể, điểm tổng kết được tính theo công thức:

Điểm tổng kết = (30% *Điểm giữa kì*) + (70% *Điểm cuối kì*)

Ngoài ra, bạn coi lại Bài 3.1 nếu chưa thành thạo việc xuất có định dạng.

BÀI TẬP

Bt 4.7.1 Làm lại Bài tập 1.1.1 và Bài tập 1.1.2 nhưng xuất ra file văn bản (thay vì màn hình).

[6] Trong trường hợp mã số phức tạp thì ta phải dùng kiểu chuỗi.

Bt 4.7.2 Viết chương trình đọc một danh sách các tên trong một tập tin văn bản (mỗi tên trên một dòng) và tạo ra mỗi file văn bản chứa bài hát "Happy birthday" riêng cho mỗi tên[7] (giống Bài tập 1.2.1).

Bt 4.7.3 Làm lại Bài tập 1.2.2 nhưng chương trình sẽ đọc nội dung đoạn trích "*có đánh dấu*"[8] từ một file văn bản, chẳng hạn:

```
<Ro> bước thơ thẩn trong vườn, tơ tưởng đến <Ju>. …
```

Trong đó `<Ro>` đánh dấu cho Roméo và `<Ju>` đánh dấu cho Juliette. Sau đó xuất ra màn hình đoạn trích với tên cho Roméo và tên cho Juliette được thay vào các vị trí đánh dấu tương ứng.

Bt 4.7.4 Làm lại Bài tập 1.7.1 và Bài tập 1.7.6 nhưng xuất ra file văn bản (thay vì màn hình)[9].

Bt 4.7.5 Như đã nói, mã của Bài 2.7 quá dông dài. Thực sự thì bản thiết kế font chữ (các nét vẽ (kí hiệu) của lưới chữ) có thể được để trong file (văn bản) mà không cần (và không nên) mã cứng trong chương trình. Bạn hãy viết lại mã của Bài 2.7 thật ngắn gọn bằng cách để lưới chữ của các kí hiệu trong file văn bản (chung trong một file hoặc mỗi chữ cái/dấu trong một file riêng) kết hợp với kĩ thuật xuất offline (Bài tập 4.2.1). Cách làm này cũng có lợi là nếu bạn muốn đổi "font chữ", bạn chỉ cần đổi các file văn bản này mà không phải viết lại chương trình. Do đó bạn có thể hợp nhất với Bài tập 2.7.2.

Bt 4.7.6 Thêm chức năng nhập/xuất từ file cho chương trình ở Bài tập 3.7.7.

Bt 4.7.7 Làm lại Bài tập 4.2.3 và Bài tập 4.2.5 nhưng xuất ra file văn bản.

Bt 4.7.8 Làm lại Bài tập 4.4.5 và Bài tập 4.4.6 nhưng cho chuỗi là toàn bộ nội dung của một file văn bản[10].

---

[7] Bạn có thể tưởng tượng đó là chương trình in thiệp mời hàng loạt từ danh sách khách mời (tương tự chức năng Mail merge của Microsoft Word).

[8] Ý tưởng đơn giản này được dùng trong rất nhiều *công nghệ xử lý động* (nội dung được thay đổi) đặc biệt là *Web động*. (Ta sẽ tìm hiểu thêm sau.)

[9] Với các "bức tranh ASCII Art" thì việc kết xuất ra file văn bản có nhiều lợi ích do không giới hạn kích thước (chiều rộng và chiều cao) của bản vẽ, hơn nữa bạn có nhiều lựa chọn về font chữ (như kích thước có thể rất nhỏ, …) và có thể lưu lại, sao chép, …. Cũng lưu ý tắt chế độ "Word Wrap" để khỏi bị "bẻ dòng" và khi xuất văn bản UTF-8 bạn cũng phải chọn font Unicode để hiển thị đúng văn bản Unicode trong chương trình xem file văn bản (như Notepad).

[10] Thực ra tần số *n-gram* chỉ có ý nghĩa trên văn bản lớn (chuỗi dài) nhất là khi *n* lớn. Bạn cũng bỏ qua không đếm các kí hiệu như dấu câu, khoảng trắng.

Bt 4.7.9 Hãy bổ sung thao tác đọc/ghi ma trận từ/vào file cho kiểu ma trận ở Bài tập 4.5.7. Các thao tác này đặc biệt có ích cho các ma trận có kích thước lớn.

Bt 4.7.10 Chương trình Char Paint trong Bài 3.9 còn thiếu hai chức năng quan trọng là lưu document ("bảng vẽ chữ") xuống file và nạp document từ file[11]. Hãy thêm hai chức năng này cho chương trình Char Paint. (*Gợi ý*: dùng kĩ thuật xuất offline với buffer (mảng 2 chiều) cho document).

[11] Hai chức năng này chính là Save và Open, có trong hầu hết các phần mềm như Notepad, Microsoft Word, MS Paint, …

BÀI 4.8

Bạn đã thấy cách tìm nghiệm đa thức bậc 2 bằng công thức nghiệm ở Bài 1.5. Các đa thức bậc 3, bậc 4 cũng có công thức nghiệm *ở dạng đóng*[1] với các phép toán cộng, trừ, nhân, chia và khai căn (nghĩa là có có công thức với số lượng cố định các phép toán trên) mặc dù các công thức đó rất phức tạp. Tuy nhiên với đa thức bậc 5 trở lên thì nói chung không có công thức nghiệm như vậy. Chẳng hạn không có công thức nghiệm cho một đa thức bậc 5 đơn giản như $p(x) = x^5 - x + 1$. Như vậy có thể nói là Toán bó tay rồi. Tuy nhiên ta vẫn có thể làm được bằng một cách rất đơn giản với sự trợ giúp của máy tính mà ta tạm gọi là *phương pháp Tin* (nghĩa là phương pháp chỉ có ý nghĩa hay khả thi hay trở nên mạnh mẽ/hiệu quả khi có sự trợ giúp của máy tính).

Bạn hãy xem lại *phương pháp chia đôi* mà ta đã dùng để tính căn bậc 2 của một số dương ở phần Mở rộng 1.5. Ta tổng quát hóa phương pháp này cho bài toán tổng quát hơn: tìm nghiệm của đa thức $f(x)$. Cụ thể ta sẽ mò nghiệm x_0 của f trong khoảng $[l, r]$ (với $l \le r$) và cố thu hẹp khoảng này cho đến khi tìm thấy nghiệm. Ta thu hẹp khoảng bằng cách xét giá trị ở giữa khoảng $m = (l + r)/2$ mà bằng cách nào đó ta biết được nghiệm sẽ nằm trong khoảng $[l, m]$ hoặc $[m, r]$ và ta sẽ tìm nghiệm trong khoảng mới đó. Bằng cách lặp lại việc này ta sẽ thu hẹp khoảng chứa nghiệm x_0 cho đến khi tìm thấy. Mỗi lần lặp thì khoảng chứa nghiệm sẽ được thu hẹp đi một nửa nên phương pháp này có tên gọi là chia đôi.

Có 2 vấn đề đặt ra, một là làm sao tìm được khoảng $[l, r]$ ban đầu có chứa nghiệm và hai là làm sao biết được nghiệm x_0 thuộc $[l, m]$ hay $[m, r]$. Do đa thức là hàm số liên tục nên cả hai vấn đề này được Toán giải quyết rất đẹp bằng định lý: nếu f là hàm liên tục (nghĩa là đồ thị không "bị hở") và nếu $f(a).f(b) < 0$ (nghĩa là $f(a)$, $f(b)$ trái dấu) thì f có nghiệm trong khoảng $[a, b]$[2]. Chẳng hạn với đa thức bậc 5 $p(x)$ ở trên ta có $p(-2) = -29$ ($(-2)^5 - (-2) + 1 = -32 + 2 + 1 = -29$) và $p(0) = 1$ ($0^5 - 0 + 1$) như vậy $p(-2)$ và $p(0)$ trái dấu nên $p(x)$ có nghiệm trong khoảng $[-2, 0]$ nên ta có thể lấy khoảng $[l, r]$ ban đầu cho một nghiệm của $p(x)$ là $[-2, 0]$. Với m là giá trị ở giữa khoảng $[l, r]$ ta thấy nếu $f(m) = 0$ thì m là nghiệm còn không nếu $f(l).f(m) < 0$ (nghĩa là $f(l)$, $f(m)$ trái dấu) thì

---

[1] Closed-form.

[2] Định lý giá trị trung gian (Intermediate Value Theorem) mà lớp 12 đã biết.

khoảng [*l*, *m*] sẽ chứa nghiệm, ngược lại thì *f*(*m*), *f*(*r*) phải trái dấu (do trước đó ta đã có *f*(*l*), *f*(*r*) trái dấu) nên khoảng [*m*, *r*] sẽ chứa nghiệm.

Hàm C sau đây hiện thực phương pháp chia đôi để tìm nghiệm của một đa thức.

Mã 4.8.1 – Hàm tìm nghiệm của đa thức *f*(*x*) bằng phương pháp chia đôi

```
double findroot(double l, double r, double eps)
{
    double m, fl, fm;
    fl = f(l);
    do
    {
        m = (l + r)/2;
        fm = f(m);

        if(fm == 0)
            return m;

        if(fl * fm < 0) // tim trong [l, m]
            r = m;
        else            // tim trong [m, r]
        {
            l = m;
            fl = fm;
        }
    }
    while ((r - l) > eps);

    return m;
}
```

Hàm `findroot` trên tìm nghiệm của đa thức *f*(*x*) trong khoảng ban đầu [`l`, `r`] với sai số `eps`[3]. Khác với sai số tương đối dùng trong Mã 1.5.6, ở đây ta dùng sai số tuyệt đối. Nghĩa là, gọi x_0 là nghiệm đúng của *f*(*x*) còn *m* là giá trị tìm được (bằng hàm trên) thì $|x_0 - m| \leq$ `eps`. Để dùng được hàm này bạn cần cài đặt hàm `double f(double x)` để tính giá trị của đa thức *f* tại *x*. Chẳng hạn với đa thức *p*(*x*) ở trên thì ta viết hàm `f` như sau.

Mã 4.8.2 – Hàm `f` cho đa thức $p(x) = x^5 - x + 1$

```
double f(double x)
{
    double x2 = x * x;
    return x*x2*x2 - x + 1;
}
```

[3] Epsilon (ε).

Không có gì nhiều cho mã trên, ta đã tính giá trị của $p(x)$ là $x^5 - x + 1$. Một mẹo nhỏ ở đây là ta tính $x^2 = x.x$ trước rồi tính $x^5 = x.x^4 = x.x^2.x^2$, tiết kiệm được một phép nhân so với cách tính bình thường.

Thêm nữa để dùng hàm `findroot` trên bạn cần tìm khoảng $[l, r]$ ban đầu chứa nghiệm (mà như đã nói $f(l)$, $f(r)$ trái dấu) và xác định sai số tuyệt đối. Chẳng hạn để tìm nghiệm của đa thức $p(x)$ trên chính xác đến chữ số 9 sau dấu phẩy thập phân (10^{-9} mà viết gọn trong C là `1e-9`) mà ta đã biết là [-2, 0] chứa nghiệm ($p(-2)$, $p(0)$ trái dấu) thì ta có thể dùng lệnh xuất như sau (sau khi đã cài hàm `f` để tính giá trị của $p(x)$ như trên):

```
printf("%.9lf\n", findroot(-2, 0, 1e-9));
```

Và ta được kết quả là `-1.167303978`. Vậy một nghiệm của đa thức $p(x) = x^5 - x + 1$ là -1.167303978. Các câu hỏi như: $p(x)$ còn nghiệm nào nữa hay không? $p(x)$ có bao nhiêu nghiệm (thực)? … sẽ không được trả lời ở đây.

Lưu ý là cài đặt ở trên rất linh hoạt (tổng quát). Bạn chỉ cần cài đặt hàm `f` cụ thể cho đa thức muốn tìm nghiệm và đưa ra khoảng chứa nghiệm ban đầu. Chẳng hạn để tính $\sqrt{2}$ nghĩa là nghiệm của đa thức $x^2 - 2$ (có nghiệm trong khoảng [0, 2]) thì bạn viết hàm `f` như sau:

```
double f(double x)
{
    return x*x - 2;
}
```

Và gọi `findroot(0, 2, 1e-9)` sẽ được giá trị gần đúng $\sqrt{2}$ là 1.414213561 (sai số 10^{-9}). Thế còn tính $\sqrt[3]{2}$ thì sao?

Phương pháp chia đôi này hay ở chỗ là nó rất đơn giản. Thật sự nếu không dựa vào khả năng tính toán của máy (lặp lại việc tính toán rất nhiều lần) thì phương pháp này chỉ là đồ bỏ đi. Đây là dạng “cần cù bù thông minh” và là phương pháp điển hình của Tin (tức là phương pháp phù hợp với máy). Quan trọng hơn, phương pháp này cực mạnh do tính tổng quát của nó. Thật ra phương pháp này không chỉ giúp tìm nghiệm của đa thức mà tìm nghiệm của hàm liên tục bất kì. Chẳng hạn ta đã biết rằng $\sin(\pi) = 0$. Do đó ta có thể tính số π bằng cách tìm nghiệm của hàm $\sin(x)$ trong khoảng $[\pi/2, 3\pi/2]$ do $\sin(x)$ còn có các nghiệm khác ngoài π[4]. Khoảng đơn giản hơn là [1, 5]. Ta viết lại hàm `f` như sau (nhớ `#include <math.h>`):

```
double f(double x)
{
    return sin(x);
}
```

Và gọi `findroot(1,5,1e-9)` sẽ được giá trị gần đúng của π là 3.141592654.

---

[4] Nghiệm của $\sin(x)$ là $x = k\pi$, $k \in Z$.

Bài học ở đây là *Toán không phải là tất cả*. Thật sự Toán rất quan trọng với Tin (và mọi ngành khoa học, kĩ thuật khác) nhưng Tin có bản chất riêng và cũng quan trọng với Toán (và mọi ngành khoa học, kĩ thuật khác). Có một ngành gọi là *giải tích số*[5] mà mục đích là dùng các phương pháp số (các phương pháp Tin tương tự như trên) để giải và nghiên cứu Toán.

Bạn đã thấy cách dùng phương pháp chia đôi để tính căn? Còn phương pháp nào hay hơn không? Xem hàm tính căn sau.

Mã 4.8.3 – Hàm tính căn bằng phương pháp Newton

```
double newtonsqrt(double y, double eps)
{
    double x0 = y/2;
    while(1)
    {
        double x1 = (x0 + y/x0)/2;
        if((x1 > x0 ? x1 - x0 : x0 - x1) < eps)
            return x1;
        x0 = x1;
    }
}
```

Hàm này tính căn bậc 2 của số thực dương y với sai số tuyệt đối không quá eps. Chẳng hạn để tính $\sqrt{2}$ gọi `newtonsqrt(2, 1e-9)`. Như tên gọi hàm này do Newton[6] viết:) Đùa thôi, hàm này viết theo *phương pháp Newton*. Nhìn mã ta thấy rằng phương pháp này cũng là *phương pháp lặp* giống như phương pháp chia đôi. Thứ 2 mã rất ngắn (thực sự thì chỉ có điều kiện của `if` là hơi rối mà thực ra đó là |x1 − x0| < eps). Điều đó cũng dẫn đến mã này rất khó hiểu. Thực sự thì phương pháp Newton rắc rối hơn phương pháp chia đôi, phương pháp này cũng không tổng quát bằng phương pháp chia đôi (nó đòi hỏi nhiều hơn ở hàm cần tìm nghiệm) nhưng bù lại nó sẽ hiệu quả hơn (nghĩa là chạy nhanh hơn). Bạn thử sửa mã 2 hàm tính căn (hàm tính căn trên bằng phương pháp Newton với hàm tính căn bằng phương pháp chia đôi) để đếm số lần lặp và qua đó đánh giá cái nào nhanh hơn. Tôi sẽ dừng chủ đề Toán cho Tin hay Tin cho Toán ở đây và không nói thêm gì nữa.

Nếu bạn chưa thấy ấn tượng với tính tổng quát của phương pháp nhị phân (chia đôi), hãy xem cách dùng phương này cho một bài toán quan trọng của Tin là *bài toán tìm kiếm*: trong một đống thông tin (hay dữ liệu) hãy tìm ra một thông tin (hay dữ liệu) mong muốn nào đó. Đây có thể nói là bài toán quan trọng nhất của Tin. Tùy theo qui mô, tùy theo cách tổ chức

[5] Numerical analysis.

[6] Isaac Newton, nhà Vật lý học và Toán học người Anh. (nhưng không là lập trình viên?:)).

mà ta có những cách làm khác nhau. Ở đây xét ngữ cảnh đơn giản nhất: tìm một phần tử trong một mảng các số. Giả sử ta có mảng *a* chứa *n* số (nguyên chẳng hạn) làm sao tìm được vị trí của phần tử có giá trị *k* cho trước nào đó. Bây chừ giả sử thêm nữa là mảng *a* đã được sắp tăng dần: tức là các phần tử có giá trị nhỏ hơn sẽ đứng trước các phần tử có giá trị lớn hơn, khi đó ta có thể dùng thuật toán chia đôi để tìm mà ta gọi là tìm kiếm chia đôi (hay tìm kiếm nhị phân): Xuất phát từ khoảng nguyên [*l*, *r*] là [0, *n* - 1] cho vị trí của phần tử cần tìm và cố thu hẹp khoảng này cho đến khi tìm thấy hoặc không tìm thấy. Ta thu hẹp khoảng bằng cách xét vị trí ở giữa khoảng *m* = (*l* + *r*)/2 mà nếu *a*[*m*] = *k* thì ta tìm thấy còn nếu *k* < *a*[*m*] thì ta tìm trong khoảng [*l*, *m* - 1], ngược lại *k* > *a*[*m*] thì ta sẽ tìm trong khoảng [*m* + 1, *r*]. Bằng cách lặp lại việc này ta sẽ thu hẹp khoảng cần tìm cho đến khi tìm thấy hoặc không (khi mã khoảng tìm là khoảng rỗng, *l* > *r*). Mỗi lần lặp thì khoảng cần tìm sẽ được thu hẹp đi một nửa nên phương pháp này có tên gọi là chia đôi và cho hiệu quả tìm cực nhanh. Chẳng hạn nếu mảng có $n = 2^{64}$ = (số cực lớn) thì chỉ cần tìm tối đa $\log_2 2^{64} = 64$ lần là ta có kết quả. Thuật toán này được cài đặt như sau.

Mã 4.8.4 – Hàm tìm kiếm nhị phân

```
int binarysearch(int a[], int l, int r, int k)
{
    while(l <= r)
    {
        int m = (l + r)/2;
        if(a[m] == k)   // tim thay
            return m;
        if(k < a[m])    // tim trong [l, m - 1]
            r = m - 1;
        else            // tim trong [m + 1, r]
            l = m + 1;
    }

    return -1;  // l > r: khong tim thay
}
```

Hàm trên sẽ tìm trong mảng a vị trí (chỉ số) của phần tử có giá trị k trong khoảng vị trí ban đầu là [l, r]. Hàm trả về chỉ số của phần tử nếu tìm thấy và trả về -1 nếu không tìm thấy (vì chỉ số từ 0 trở đi nên -1 không là chỉ số do đó nó được dùng với ý nghĩa là không có). Sau khi đã có mảng a với n phần tử đã được sắp (chẳng hạn đã nhập dữ liệu cho a từ người dùng hay từ file và sắp xếp tăng dần) và có giá trị cần tìm k thì ta có thể gọi: `binarysearch(a, 0, n - 1, k)` để tìm trong toàn bộ mảng. Nếu đã rành đệ qui thì hàm tìm nhị phân trên có thể viết theo kiểu đệ qui như sau.

Mã 4.8.5 – Hàm tìm kiếm nhị phân viết theo kiểu đệ qui

```
int binarysearch(int a[], int l, int r, int k)
{
    if(l > r)       // khong tim thay
        return -1;

    int m = (l + r)/2;
    if(a[m] == k)   // tim thay
        return m;
    if(k < a[m])    // tim trong [l, m - 1]
        return binarysearch(a, l, m - 1, k);
    else            // tim trong [m + 1, r]
        return binarysearch(a, m + 1, r, k);
}
```

Lưu ý là để có thể dùng tìm kiếm nhị phân thì dữ liệu đã phải được tổ chức rất tốt: mảng phải được sắp xếp tăng dần. Như vậy một bài toán quan trọng khác trong Tin là tổ chức dữ liệu cho tốt mà quan trọng là *sắp xếp*. Cũng vậy tùy theo qui mô, tùy theo kiểu dữ liệu, thứ tự sắp xếp, bộ nhớ trong ngoài, … mà ta có những cách làm khác nhau. Ở đây xét ngữ cảnh đơn giản nhất: sắp xếp tăng dần một mảng các số. Bạn đã thấy một cách làm "mang tính Toán" trong Bài 4.1 hay Bài 4.3 dựa trên khái niệm nghịch thế. Nhưng đó chưa phải là cách sắp xếp tốt nhất. Có 101 cách sắp xếp khác nhau.

Trong Bài 4.1 ta đã giải bài toán tìm số lớn thứ *k* trong các số mà người dùng đã nhập. Ta đã thấy cách làm online và offline. Ta đã nói rằng nếu *k* cố định và nhỏ (như *k* = 1 hay *k* = 2) thì ta nên làm online và không cần trữ dữ liệu lại còn nếu *k* không cố định hay *k* lớn thì ta nên làm offline bằng cách trữ dữ liệu lại (trong mảng) sau đó sắp xếp rồi truy cập kết quả. Có cách nào làm online khi *k* không cố định hay *k* lớn không? Có: ta cũng sẽ dùng mảng để lưu trữ nhưng ta sẽ không đợi nhập hết rồi mới sắp xếp mà ta sẽ vừa cho nhập vừa sắp xếp online. Điều đó có nghĩa là "nhập tới đâu sắp xếp tới đó": đầu tiên mảng rỗng, sau đó nhập một số thì ta đưa vào mảng và sắp xếp (mà thực ra là chỉ đưa vào thôi vì mảng 1 phần tử thì hiển nhiên là đã sắp), sau đó nhập một số nữa thì ta đưa vào mảng và sắp xếp, … Như vậy khi nhập phần tử mới thì ta có mảng chứa các phần tử trước đó đã được sắp. Ta cần thêm phần tử mới vào và sắp xếp lại. Như vậy với cách làm này ta luôn có kết quả sắp xếp khi nhập lưng chừng và dễ dàng truy cập được phần tử lớn thứ *k* (tính cho đến lúc nhập). Trong ngữ cảnh này thì thuật toán sắp xếp phù hợp nhất là *sắp xếp chèn*. Đây là cách mà những người chơi bài tiến lên hay làm[7]: chia 12 cây bài, bốc lần lượt từng cây, bốc tới đâu xếp tới đó[8]. Vậy cách làm tự nhiên và hiệu quả nhất là: trên tay đang có các quân bài đã

[7] Tài liệu này không khuyến khích bạn chơi bài nha:)

[8] Ngược lại là đợi chia hết 12 cây rồi bốc và sắp một lượt.

sắp, ta tìm cách chèn quân bài mới vào đúng vị trí để được sắp. Mảng *a* với các phần tử từ 0 đến *t* đã sắp ta cần tìm vị trí chèn phần tử mới *x* để được mảng sắp với các phần tử 0 đến *t* + 1. Nếu là sắp giảm dần thì ta có thể làm như sau: đi từ cuối mảng và dời các phần tử ra sau khi phần tử chèn vào có giá trị lớn hơn, nếu không thì ta đã tìm thấy vị trí cần chèn, chèn vào và dừng. Hàm sau cài đặt thuật toán này.

Mã 4.8.6 – Hàm chèn phần tử mới để sắp xếp chèn

```
void insert(int a[], int &n, int x)
{
    int i = n - 1;
    for(; i >= 0 && a[i] < x; i--)
        a[i + 1] = a[i];
    a[i + 1] = x;
    n++;
}
```

Bạn có thể ráp vào chương trình hoàn chỉnh như sau. Ở đây để thực tế hơn ta sẽ không yêu cầu người dùng cho biết trước số lượng sẽ xử lý (đây là tình huống thường gặp của xử lý online, ta không biết trước lượng dữ liệu sẽ xử lý, nhận dữ liệu tới đâu xử lý tới đó). Bạn chạy chương trình để hiểu rõ hơn nhé.

Mã 4.8.7 – Chương trình tìm số lớn thứ *k* bằng cách làm online

```
int main()
{
    int a[1000];
    int n = 0;
    while(1)
    {
        int cmd, x, m;
        printf("Ban muon (1 - Nhap so, 2 - Tra so, 3 -
Dung): ");
        scanf("%d", &cmd);
        if(cmd == 1)
        {
            printf("Nhap so nguyen: ");
            scanf("%d", &x);
            insert(a, n, x);
        }
        else if(cmd == 2)
        {
            printf("Ban muon tim so lon thu may: ");
            scanf("%d", &m);
            if(1 <= m && m <= n)
                printf("So lon thu %d la: %d\n", m,
                        a[m - 1]);
```

```
            else
                printf("Vi thu khong hop le\n");
        }
        else
        {
            printf("Bye!");
            break;
        }
    }

    return 0;
}
```

Thật sự là có rất nhiều thuật toán sắp xếp hay hơn, hiệu quả hơn (và phức tạp hơn) thuật toán sắp xếp chèn. Tuy nhiên trong ngữ cảnh này, có lẽ thuật toán sắp xếp chèn là tự nhiên nhất, phù hợp nhất, tốt nhất[9]. Bài học ở đây là có nhiều phương pháp (cách thức, thuật toán) khác nhau để giải một bài toán. Tất cả chúng đều có giá trị và không có cái tệ nhất cũng không có cái tốt nhất. Chúng đều có điểm mạnh (hay, thuận lợi) và điểm yếu (dở, hạn chế) trong các tình huống, ngữ cảnh khác nhau. Vấn đề là ta phải vận dụng cho khôn khéo, phù hợp. Tôi cũng dừng chủ đề này ở đây và không nói thêm gì nữa.

Ở trên bạn đã tính được số π đến 8 chữ số sau dấu chấm thập phân là: 3.14159265[10]. Còn có cách tính số π nào không? Còn rất nhiều. Cách trên là cách tệ so với nhiều cách khác. Ở đây tôi giới thiệu một cách … còn tệ hơn nữa:) Nhưng bạn biết rồi đó, chơi chữ thôi, tệ hay tốt là theo tiêu chí gì. Theo cách nào đó đây là thuật toán rất tệ nhưng theo cách nào đó đây là thuật toán tốt nhất: *thuật toán Monte Carlo* (hay đúng hơn là *phương pháp*[11] *Monte Carlo*).

Xét một thí nghiệm như sau: gieo một điểm ngẫu nhiên trên hình vuông đơn vị (hình vuông [0, 1]x[0, 1]) và ghi nhận điểm đó có nằm trong góc phần tư hình tròn đơn vị (hình tròn bán kính 1) như Hình 4.8.1. Do điểm gieo được là ngẫu nhiên (mà theo thuật ngữ xác suất thì nó có phân phối đều) nên ta có xác suất để điểm rơi vào góc phần tư hình tròn đơn vị là:

$$\begin{aligned} P &= (\text{Diện tích góc tư hình tròn})/(\text{Diện tích hình vuông}) \\ &= (\pi 1^2/4)/(1^2) = \pi/4 \end{aligned}$$

[9] Thật ra có thể dùng các cấu trúc *cây được sắp* như Heap sẽ cho cách làm hiệu quả hơn nhưng ta sẽ không đi xa như vậy.

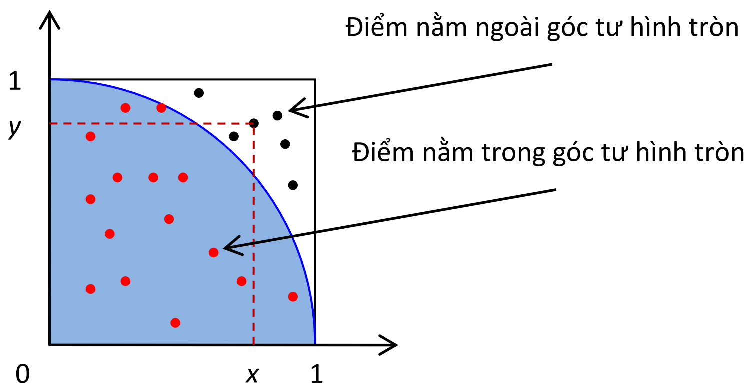
[10] Kết quả trên có sai số là 10^{-9} nên chữ số cuối cùng có thể làm tròn nên ta không biết.

[11] Thuật ngữ phương pháp thường được dùng theo nghĩa tổng quát hơn thuật ngữ thuật toán.

Giả sử ta lặp lại thí nghiệm này n lần độc lập (các lần không liên quan gì nhau) và đếm số lần mà điểm rơi vào góc phần tư là m lần. Khi đó tần suất điểm rơi vào là: $f = m/n$. Bây chừ nếu n lớn (tức là số lần lặp lại thí nghiệm lớn) thì tần số sẽ xấp xỉ xác suất tức là $f \approx P$ nên $m/n \approx \pi/4$ nên $\pi \approx 4m/n$. Và đây là cách mà ta dùng để tính số π.

Đây là kết quả của cái gọi là *luật số lớn* trong lý thuyết xác suất mà có ý nghĩa rất lớn trong lý thuyết. Còn trong thực tế nó chẳng thực tế chút nào. Bạn hãy xem ta vận dúng nó để tính số π nhé. Bạn cắt một miếng bìa hình vuông (cạnh 1), tô góc phần tư hình tròn như hình. Lấy một viên phấn (thiệt nhỏ) gieo ngẫu nhiên (mà đại khái là nhắm mắt ném đại) vào miếng bìa rồi nhìn xem viên phấn có rơi vào góc phần tư hình tròn không. Bạn làm lại rất nhiều lần như vậy, chẳng hạn 1000 lần, 10000 lần (n = 10000, 10000) và phải đảm bảo là các lần độc lập nhau. Ghi nhận số lần rơi vào góc phần tư m. Từ đó tính được π (bằng $4m/n$). Không thực tế vì: (1) bạn đảm bảo gieo ngẫu nhiên vào miếng bìa không? (thậm chí bạn thường ném nó ra ngoài miếng bìa), (2) bạn có đủ sức làm 1000 lần như vậy không? (tôi cá rằng bạn không đủ kiên nhẫn để làm[12]), (3) bạn có đảm bảo các lần độc lập nhau không? (hầu như không thể, do cùng một người gieo nên dù bạn có cố ý ném đại thì dường như bạn cũng gieo cùng một cách (cách của bạn, theo cơ địa của bạn), để đảm bảo điều này thì có lẽ nên mời 1000 người khác nhau để làm). Tóm lại là không thực tế. Đây cũng là khó khăn chung của các phương pháp dựa trên xác suất, thống kê như vậy.

Hình 4.8.1 – Thí nghiệm gieo điểm



Nhưng bây giờ, bạn không phải là người tiến hành thí nghiệm trên mà là máy, máy sẽ tiến hành *thí nghiệm ảo* hay *mô phỏng* thí nghiệm trên bằng chương trình thì cả 3 vấn đề trên đều giải quyết được. Quan trọng là làm

[12] Nghe đồn một số nhà thống kê như Mendel đã làm được việc này mà còn khó hơn đó là trồng đậu và cũng được đền đáp khi phám ra các định luật Mendel.

sao tạo được điểm ngẫu nhiên trong hình vuông, đó là việc tạo cặp số (*x*, *y*) với *x*, *y* ngẫu nhiên (hay phân bố đều) trên [0, 1]. C có thể giúp ta làm việc này bằng hàm tạo số ngẫu nhiên mà bạn đã biết trong phần Mở rộng 2.6 (Mã 2.6.4). Như vậy ta có thể viết hàm tính số π bằng cách mô phỏng thí nghiệm trên mà ta gọi là phương pháp Monte-Carlo.

Mã 4.8.8 – Hàm tính số π bằng phương pháp Monte-Carlo

```
double montecarlopi(int n)
{
    int m = 0;
    srand(time(NULL));
    for(int i = 1; i <= n; i++)
    {
        double x = (double)rand()/RAND_MAX;
        double y = (double)rand()/RAND_MAX;
        if(x*x + y*y <= 1)
            m++;
    }

    return 4.0*m/n;
}
```

Không có gì nhiều ngoài việc làm sao "nhìn xem" điểm (*x*, *y*) có nằm trong góc phần tư hình tròn không. Cách nhìn của máy (hay đúng hơn là nhìn bằng Toán) nè: đo khoảng cách từ điểm (*x*, *y*) đến tâm hình tròn (0, 0) và nếu khoảng cách không lớn hơn bán kính hình tròn thì nó nằm trong. Khoảng cách là $\sqrt{x^2+y^2}$ và bán kính là 1 như vậy điều kiện nằm trong là: $\sqrt{x^2+y^2} \le 1$. Tuy nhiên trên máy thì ta làm thêm chút nữa, điều kiện trên tương đương với $\sqrt{x^2+y^2}^2 \le 1^2$ tương đương với $x^2+y^2 \le 1$. Với Toán thì các điều kiện trên là như nhau nhưng với Tin (máy) thì khác nhau hoàn toàn, điều kiện sau cùng không dùng phép toán lấy căn (mà phép toán này tốn kém thế nào các bạn đã biết rồi đấy).

Bạn cài và chạy thử với các giá trị của *n* (số lần thực hiện thí nghiệm), chẳng hạn gọi `montecarlopi(1000)` cho *n* = 1000. Và quả thực kết quả ra rất tệ, của tôi ra 3.22[13]. Thậm chí đúng mỗi chữ số 3. Tăng *n* lên bạn sẽ có kết quả chính xác hơn (chút xíu), chẳng hạn *n* = 10000 được 3.1168 (thêm được chữ số), *n* = 1000000 được 3.14253 (được thêm chữ số nữa)! Rõ ràng đây là phương pháp rất rất tệ. Thật sự chẳng ai dùng phương pháp này để tính π: nó xấp xỉ quá chậm (tốn nhiều lần lặp *n* nhưng sai số giảm chậm) và ta cũng không kiểm soát được sai số (sai số của các kết quả trên là bao nhiêu? cần *n* bằng bao nhiêu để chẳng hạn đúng được 4 chữ số sau dấu thập phân? những câu hỏi không dễ trả lời). Tuy nhiên đây là phương pháp

[13] Hiển nhiên mỗi lần chạy kết quả mỗi khác do ngẫu nhiên mà.

tuyệt vời. Nó rất tổng quát và cực dễ để thực hiện. Nếu như lý thuyết xác suất giúp dùng tất yếu để mô hình ngẫu nhiên thì các phương pháp như Monte Carlo trên máy lại dùng ngẫu nhiên để giải tất yếu. Toàn bộ phương pháp trên cậy nhờ việc phát sinh số ngẫu nhiêu (phân bố đều) trên khoảng [0, 1] mà ở đây là hàm rand.

Thật sự nhờ các thuật toán mô phỏng như thế này mà máy tính giúp cho lý thuyết xác suất thống kê trở nên thực tế và ứng dụng được nhiều hơn. Hơn nữa xu thế dùng lý thuyết xác suất cho tất cả các ngành khác nhau kể cả Tin học ngày càng nhiều và nhóm các thuật toán dùng xác suất thống kê này rất quan trọng mà ta sẽ tìm hiểu sâu ở tài liệu khác.

Ngoài Toán với ngành Giải tích số, Tin cũng quan trọng cho các ngành khoa học khác như Sinh (với Sinh học tính toán, Sinh Tin), Hóa (với Hóa Tin), Lý (với Lý Tin), … Tin cũng quan trọng cho các ngành khoa học xã hội như ngôn ngữ với ngôn ngữ học tính toán (NLP), các ngành kinh tế, quản trị, … mà ta nghe chung là *số hóa*. Tất cả đều dựa vào máy tính mà chủ yếu là 2 thứ: khả năng tổ chức, quản lý dữ liệu (mà chủ yếu dựa trên lượng bộ nhớ khổng lồ) và các phương pháp xử lý kiểu Tin (mà chủ yếu dựa trên khả năng tính toán vô hạn, không mệt mỏi).

MỞ RỘNG 4.8 – Thuật toán tìm kiếm quay lui và bài toán 8-hậu

Ta đã nói rằng tìm kiếm (và sắp xếp) là bài toán rất quan trọng và tổng quát trên máy. Thực sự thì nó cũng quan trọng và tổng quát nói chung. Thật vậy có thể nói rằng hầu như mọi bài toán đều là *bài toán tìm kiếm*. Ta tìm output từ input hay tìm lời giải từ yêu cầu của bài toán. Ta có thể phát biểu chung bài toán này như sau:

- *Input*: tập *U*, tính chất *P* trên *U* (nghĩa là *P* là vị từ trên *U*, nghĩa là *P*(*x*) hoặc đúng hoặc sai với mọi *x* thuộc *U*. Khi *P*(*x*) là đúng ta nói *x* thỏa tính chất *P*, ngược lại thì *x* không thỏa tính chất *P*).
- *Output*: phần tử x_0 thuộc *U* thỏa tính chất *P* (nghĩa là $P(x_0)$ đúng).

Bài toán trên là bài toán tìm kiếm, *U* gọi là không gian tìm kiếm, các phần tử của *U* gọi là các ứng viên, tính chất *P* gọi là tiêu chí tìm kiếm, phần tử x_0 (thỏa tính chất *P*) gọi là lời giải.

Chẳng hạn với bài toán tìm nghiệm của đa thức $p(x) = x^5 - x + 1$ thì $U = R$, nghĩa là các ứng viên là các số thực, tiêu chí tìm kiếm là $P(x)$: $x^5 - x + 1 = 0$, lời giải chính là nghiệm của đa thức *p*(*x*) (mà một lời giải "gần đúng" mà ta đã tìm thấy ở trên là -1.167303978).

Bài toán tìm phần tử trong mảng cũng là bài toán tìm kiếm, *U* là tập các số nguyên không âm (tập chỉ số), tiêu chí tìm kiếm là *P*(*i*): *a*[*i*] = *k*, lời giải chính là vị trí tìm thấy phần tử có giá trị *k* trong mảng mà ta đã qui ước -1 nghĩa là "không tìm thấy" do -1 không là ứng viên.

Chiến lược để giải bài toán tìm kiếm gọi là *phương pháp tìm kiếm*.

Một phương pháp tìm kiếm rất phổ biến là phương pháp *tìm kiếm vét cạn* dựa trên nguyên lý Edison: “Muốn tìm được cây kim trong đống rơm, hãy lần lượt rút từng cộng rơm cho đến khi rút được cây kim”. Phương pháp này rất tổng quát (có thể dùng giải hầu hết mọi bài toán tìm kiếm) vì nó không đòi hỏi nhiều trên input. Đặc điểm của phương pháp này là: xét lần lượt các ứng viên, kiểm tra tiêu chí tìm kiếm trên tất cả các ứng viên để không bỏ sót lời giải. Một kiểu “cần cù bù thông minh”, rất phù hợp với máy tính.

Thuật toán để giải bài toán tìm kiếm gọi là thuật toán tìm kiếm. Chẳng hạn với chiến lược tìm kiếm vét cạn thì thuật toán “ngây thơ” nhất để làm đó là tìm kiếm tuần tự: xét lần lượt các ứng viên, kiểm tra tiêu chí tìm kiếm trên tất cả các ứng viên. Chẳng hạn với bài toán tìm kiếm trên mảng ta cài đặt thuật toán này như sau.

Mã 4.8.9 – Thuật toán tìm kiếm tuần tự trên mảng

```
int linear_search(int a[], int n, int k)
{
    for(int i = 0; i < n; i++)
        if(a[i] == k)
            return i;
    return -1;
}
```

Thuật toán này hầu như luôn áp dụng được cho phần lớn bài toán tìm kiếm nhưng bù lại thời gian thực hiện rất lâu (tỉ lệ với kích thước không gian tìm kiếm do có thể phải thử hết ứng viên, ở trên là tỉ lệ với số phần tử trong mảng, n). Trong một số trường hợp thì đây là thuật toán tìm kiếm duy nhất dùng được. Tuy nhiên một số trường hợp khác thì cách kiểm tra không gian tìm kiếm khôn khéo hơn sẽ cho thời gian nhanh hơn. Chẳng hạn ta đã dùng thuật toán tìm kiếm nhị phân (hay chia đôi) để giải 2 bài toán tìm kiếm trên (trong đó ở bài toán đầu, ta chấp nhận tìm “gần đúng”). Ta cũng đã biết rằng thuật toán này rất hiệu quả, cho thời gian rất nhanh. Tuy nhiên để có thể áp dụng thuật toán này thì đòi hỏi không gian tìm kiếm phải được “sắp xếp” hay tổ chức tốt mà không phải lúc nào ta cũng có được điều này.

Một trường hợp khác mà ta có thể tăng thời gian tìm kiếm do khôn khéo loại các ứng viên là thuật toán *tìm kiếm quay lui* mà bài toán 8-hậu là một trường hợp áp dụng điển hình. Bài toán 8-hậu: “đặt 8 quân hậu vào bàn cờ sao cho các quân hậu không tấn công lẫn nhau”. Phân tích sơ lược ta có 8 quân hậu giống cần đặt vào 8 ô trong 8×8 = 64 ô của bàn cờ. Như vậy ta có tất cả $C^8_{64} = 64!/(56!8!) \approx 10^9$ ứng viên cần kiểm tra. Phân tích sâu hơn một chút ta thấy 8 quân hậu cần để trên các cột khác nhau ta gọi là hậu 1, 2, ..., 8 mà hậu *k* là ở cột *k* hơn nữa các hậu phải ở trên các

dòng khách nhau nên ta cần tìm bộ (h_1, h_2, …, h_8) trong các hoán vị của {1, 2, …, 8} (8 dòng) sao cho quân hậu ở cột i, dòng h_i không cùng đường chéo với quân hậu ở cột j, dòng h_j ($1 \le i \ne j \le 8$). Như vậy số ứng viên là: 8! = 40320. Mặc dù ta có thể tìm kiếm tuần tự do con số 40320 ứng viên không phải là lớn trên máy[14]. Trước hết ta không có cách nào tìm nhị phân trên này do đó không thể loại bỏ một nửa số ứng viên mỗi lần kiểm tra như tìm kiếm nhị phân nhưng ta có thể làm nhanh hơn với nhận xét:

- Nếu $h_1 = 1$, $h_2 = 2$ thì (h_1, h_2, h_3, …, h_8) không là lời giải với mọi h_3, …, h_8. Có 6! = 720 ứng viên như vậy.
- Nếu $h_1 = 1$, $h_2 = 3$, $h_3 = 2$ thì (h_1, h_2, h_3, h_4, …, h_8) không là lời giải với mọi h_4, …, h_8. Có 5! = 120 ứng viên như vậy.
- …

Như vậy bằng cách xây dựng lời giải dần từng bước: Bước 1: chọn h_1, Bước 2: chọn h_2, …, Bước 8: chọn h_8, ta có thể dừng sớm tại bước mà chắc chắn những bước còn lại không dẫn đến lời giải nhờ đó tăng tốc tìm kiếm do loại bỏ được nhiều ứng viên không cần xét. Thuật toán tìm kiếm này được gọi là quay lui là một áp dụng của vét cạn khi các ứng viên (và do đó lời giải) có cấu trúc. Đặc điểm của nó là:

- Xây dựng lời giải từng bước.
- Xây dựng bước tiếp theo nếu phần đang xây dựng có thể dẫn đến lời giải.
- Không xây dựng tiếp lời giải khi phần đang xây dựng không thể dẫn đến lời giải.
- Quay lui thực hiện lựa chọn khác tại bước trước nếu không còn lựa chọn nào cho bước hiện tại.

Ví dụ vài bước hoạt động của thuật toán với bài toán 8-hậu:

- $h_1 = 1$, $h_2 = 2$: không thỏa điều kiện, dừng hướng này.
- $h_1 = 1$, $h_2 = 3$: chọn được $h_3 = 5$, chọn được $h_4 = 2$, chọn được $h_5 = 4$:
  - Không chọn được h_6, quay lui chọn h_5 khác, được $h_5 = 8$
- $h_1 = 1$, $h_2 = 3$, $h_3 = 5$, $h_4 = 2$, $h_5 = 8$:
  - không chọn được h_6, quay lui chọn h_5 khác, hết cách chọn h_5, quay lui chọn h_4 khác được $h_4 = 7$.
- …

Sau đây là cài đặt bằng đệ qui thuật toán tìm kiếm quay lui này.

Mã 4.8.10 – Thuật toán tìm kiếm quay lui giải bài toán 8-hậu

```
int queens(int h[], int &k)
{
```

[14] Trường hợp tổng quát là n-hậu thì số ứng viên n! là rất lớn với n khoảng 20 trở lên, do đó ta không thể dùng thuật toán tìm kiếm tuần tự do thời gian quá lâu.

```
    if(check(h, k))
    {
        if(k == 8)
            return 1;
        k++;
        h[k] = 1;
        return queens(h, k);
    }
    else
    {
        if(h[k] > 8)
            k--;
        if(k < 1)
            return 0;
        h[k]++;
        return queens(h, k);
    }
}
```

Hàm sẽ trả về đúng (1) khi tìm thấy lời giải (để trong vector h) và sai (0) khi không tìm thấy lời giải. Trong đó hàm check kiểm tra việc đặt các quân hậu (như trong vector h) có được phép không (8 quân hậu có không tấn công nhau không) và được cài như sau.

Mã 4.8.11 – Hàm kiểm tra vị trí đặt các quân hậu có hợp lệ không

```
int check(int h[], int k)
{
    if(h[k] < 1 || h[k] > 8)
        return 0;

    for(int i = 1; i < k; i++)
        if(h[k] == h[i] || h[k] + k == h[i] + i
                || h[k] - k == h[i] - i)
            return 0;

    return 1;
}
```

Để dùng hàm tìm kiếm trên ta cần tạo vector h với h_1 khởi động là 1. Chẳng hạn hàm main viết như sau:

Mã 4.8.12 – Hàm main tìm một lời giải

```
int main()
{
    int h[9], k = 1;
    h[1] = 1;
    if(queens(h, k))
```

```
        printsol(h, k);

    return 0;
}
```

sẽ tìm lời giải đầu tiên. Hàm `printsol` dùng để xuất ra lời giải (vector h), chẳng hạn:

Mã 4.8.13 – Hàm xuất ra lời giải

```
void printsol(int h[], int k)
{
    printf("(");
    for(int i = 1; i <= k; i++)
        printf("%s%d", (i == 1 ? "" : ", "), h[i]);
    printf(")");
}
```

tìm được lời giải đầu tiên là: (1, 5, 8, 6, 3, 7, 2, 4).

Bạn có thể sửa lại hàm `main` để tìm tất cả các lời giải không. Có 92 lời giải (kể cả các lời giải đối xứng). Đáp án:

Mã 4.8.14 – Hàm `main` tìm tất cả lời giải

```
int main()
{
    int h[9], k = 1, n = 1;
        h[1] = 1;
    while(queens(h, k))
    {
        printf("Sol#%d: ", n++);
        printsol(h, k);
        printf("\n");
        h[k]++;
    }

    return 0;
}
```

Bạn đã thấy mã C giải bài toán này (thậm chí bài toán tổng quát *n*-hậu) ở Mã 1.6.3 trong Bài 1.6. Thật sự kinh ngạc về một đoạn mã siêu ngắn như vậy.

BÀI TẬP

Bt 4.8.1 Thuật toán không phải là tất cả của việc lập trình. Đối với các ứng dụng thương mại, thậm chí nó có thể không là phần chính. Tuy nhiên thuật toán là nền tảng để bạn tiến xa trong việc lập trình bởi nó quyết định tư duy lập trình, khả năng sáng tạo và giải quyết vấn đề. Điều đó có nghĩa là, không sớm thì muộn, bạn phải học, luyện và nắm vững về thuật toán. Điều đó

không có nghĩa là bạn phải có khả năng nghĩ ra các thuật toán mới hay nắm vững các thuật toán cao siêu. Tuy nhiên bạn phải thông thạo các thuật toán cơ bản nhất. Đó sẽ là một phần vốn liếng quan trọng trong nghề lập trình (cùng với các kĩ năng đã nói khác). Như cách người chơi đàn học thuộc những bản nhạc cơ bản nhất để làm vốn tiết mục (repertoire) và từ từ mở rộng nó. Bạn hãy xem lại và nắm vững các thuật toán[15] đã gặp cho đến giờ:

- Phương pháp Horner (Mở rộng 1.4, Bài tập 4.1.1)
- Phương pháp Newton (Bài tập 1.4.3, Bài tập 3.8.5, Bài 4.8)
- Phương pháp chia đôi (Mở rộng 1.5, Bài 4.8)
- Kiểm tra tính nguyên tố (Bài tập 2.5.4, Bài 3.8, Bài tập 3.8.2, Bài tập 4.1.2)
- Phân tích thừa số nguyên tố (Bài tập 2.5.5, Bài tập 4.1.2)
- Phương pháp vét cạn (Bài 3.8, Bài tập 3.8.3)
- Phương pháp Secant (Bài tập 3.8.6)
- Các thuật toán sắp xếp đơn giản (Bài tập 4.1.8, Bài tập 4.6.10)
- Thuật toán tìm chuỗi con chung dài nhất (Bài tập 4.4.9)
- Thuật toán Euclid (Mở rộng 4.5)
- Bài toán tháp Hà Nội với đệ qui và "*cấu trúc dữ liệu*" Stack (Mở rộng 4.6)
- Thuật toán tìm kiếm nhị phân (Bài 4.8)
- Phương pháp Monte-Carlo (Bài 4.8)
- Thuật toán tìm kiếm quay lui (Mở rộng 4.8)

Bt 4.8.2 Tìm hiểu thuật toán chia đa thức (được thương, dư) và ứng dụng cho Bài tập 4.5.3.

Bt 4.8.3 Tìm hiểu các thuật toán số học (cộng, trừ, nhân, chia) và ứng dụng cho Bài tập 4.5.4.

Bt 4.8.4 Tìm hiểu các thuật toán liên quan đến ma trận (nhân hai ma trận, tìm ma trận nghịch đảo, giải hệ phương trình tuyến tính, tính định thức ma trận) và ứng dụng cho Bài tập 4.5.7.

[15] Như đã nói, thuật toán, dùng theo nghĩa rộng, ám chỉ mọi cách làm. Ở đây, theo nghĩa hẹp hơn, thuật toán chỉ những cách làm phải suy tính chứ không phải những cách làm tự phát, tầm thường hay ngây thơ (naïve). Tuy nhiên, ranh giới giữa chúng không thực sự rõ ràng.

BÀI 4.9

Đây là bài học cuối cùng của Quyển I – Bí kíp Lập trình C. Trong bài này ta sẽ vận dụng nhiều công cụ và kĩ thuật đã học để viết một game rất nổi tiếng là Tetris, còn được gọi là game "Xếp gạch". Đây là game ra đời từ rất sớm (khoảng 1986) trên rất nhiều hệ máy khác nhau, phổ biến ở nhiều nước và làm say mê nhiều game thủ qua nhiều thế hệ:)

Như là cách để xác định bài toán tức là xác định yêu cầu của phần mềm, bạn hãy bắt đầu bằng việc tìm hiểu game này. Nguồn đầu tiên có thể là bài viết Tetris trên trang Wikipedia[1]. Nơi mà bạn biết rằng Tetris có nguồn gốc từ nước Nga và là game được yêu thích nhất mọi thời đại. Bạn cũng sẽ biết các thuật ngữ game như Tetriminos, Matrix, … hay các vấn đề trong game như Gravity, Rotation, … hay các tác động tích cực của game lên hoạt động của não. Để là người viết game bạn phải là người chơi game (giỏi) đã. Như vậy, bước tiếp theo bạn hãy chơi để trải nghiệm game này[2]. Tuy nhiên để là người viết game (ở đây ta không nói là sáng tạo game vì game này đã có, ta chỉ viết theo những qui tắc, … đã có). Để làm điều này ta sẽ cần tìm hiểu và nắm rõ những điều sâu hơn. Ở đây ta sẽ dùng tài liệu "Tetris Guideline" để tham khảo[3]. Ta cũng sẽ gọi chương trình/ứng dụng/phần mềm/game này là Tetris.

Bước phân tích và thiết kế giúp ta đưa ra sơ đồ module của chương trình như Hình 4.9.1. Bạn xem lại Bài 3.7 và Bài 3.9 nếu chưa rõ về module và sơ đồ module.

Đầu tiên, chương trình của ta cũng có dùng các đặc trưng cấp thấp của hệ điều hành đó là bàn phím và màn hình (cửa sổ Console). Ở đây ta sẽ dùng lại module Keyboard và module Screen ở Bài 3.9. Đây cũng là ví dụ cho thấy khả năng dùng lại khi chương trình được tổ chức theo module. Thật sự thì ta không cần "bắt" quá nhiều phím như ở Bài 3.9, nghĩa là module Keyboard cung cấp hơi nhiều so với nhu cầu của ta. Tuy nhiên ta sẽ dùng lại nguyên si mà không xóa bớt. Module Screen thì khác, nó thiếu một chức năng (dịch vụ) quan trọng mà ta cần đó là ẩn con chạy. Khi chơi game ta không muốn con chạy cứ nhấp nháy ở đâu đó vì nó làm phân tán sự chú ý nên ta cần ẩn nó đi. Như vậy ta sẽ mở rộng module Screen một chút bằng

[1] https://en.wikipedia.org/wiki/Tetris

[2] Game này có thể được chơi miễn phí ở nhiều nguồn. Chẳng hạn ở trang chủ của Tetris: http://www.testris.com

[3] Có thể xem online tại: http://tetris.wikia.com/wiki/Tetris_Guideline

cách bổ sung thêm một hàm public là `hidecursorscr` để ẩn con chạy. Hàm này viết như sau.

Mã 4.9.1 – Hàm `hidecursorscr` (của module Screen) để ẩn con chạy

```
void hidecursorscr()
{
    CONSOLE_CURSOR_INFO cursor;
    cursor.dwSize = 1; cursor.bVisible = 0;
    SetConsoleCursorInfo(_hscr, &cursor);
}
```

`CONSOLE_CURSOR_INFO` là kiểu cấu trúc được khai báo trong `wincon.h` (được `#include` kèm với `windows.h`) như sau[4]:

```
typedef struct _CONSOLE_CURSOR_INFO {
    DWORD dwSize;
    WINBOOL bVisible;
  } CONSOLE_CURSOR_INFO;
```

Kiểu này mô tả thông tin con chạy với thành phần `dwSize` xác định kích thước con chạy và `bVisible` xác định việc con chạy có được hiển thị hay không. Bạn hãy xem mã nguồn đính kèm của bài này và tra cứu để hiểu rõ nhé. Đến tầng này bạn cũng nên tò mò, khám phá để hiểu rõ hơn nhiều thứ vì bạn đã đủ công lực để hiểu gần hết. Chẳng hạn `DWORD` là kiểu được định nghĩa là: `typedef unsigned __LONG32 DWORD;` còn `__LONG32` được `#define` là: `#define __LONG32 long`. Như vậy `DWORD` chẳng qua là kiểu số nguyên `long`. Còn `WINBOOL` là kiểu được định nghĩa là: `typedef int WINBOOL;`. Như vậy `WINBOOL` chẳng qua là kiểu số nguyên `int`.

Thật sự thì module Screen đã giúp ta làm việc với màn hình một cách độc lập với thiết bị (hay đúng hơn là hệ điều hành). Tuy nhiên dịch vụ nó mang lại ở mức rất thấp. Chẳng hạn việc gọi hàm `putcharscr` sẽ giúp hiển thị một kí tự ra màn hình nhưng kí tự đó sẽ được hiển thị ngay sau khi gọi. Trong game này ta cũng sẽ hiển thị (và cập nhật liên tục) các hình ảnh đồ họa (là các kí tự) ra màn hình. Vì "khung hình" hiển thị có nhiều thành phần đồ họa nên để tránh mạnh ai nấy hiển thị ta sẽ "đồng bộ hóa" việc hiển thị bằng cách dùng kĩ thuật dùng bộ đệm hiển thị. Đây hầu như là kĩ thuật bắt buộc khi ta làm việc với hình ảnh đồ họa động (xem lại Bài 1.7). Bạn đã nghe nói về phim 24 khung hình trên giây (24 fps). Giả sử ta muốn hiển thị một đoạn "video" dài 1 giây cảnh mèo Tom đuổi bắt chuột Jerry. Với 24 fps ta sẽ hiển thị 24 khung hình (ảnh tĩnh) lần lượt vẽ. Nếu 24 khung hình này được hiển thị nhanh hơn trong 0.5s giây chẳng hạn thì ta sẽ thấy tốc độ đuổi bắt nhanh hơn (kiểu như thời gian trôi nhanh hơn) còn như 24 khung hình này được hiển thị chậm hơn, trong 5 giây chẳng hạn thì ta sẽ có hình ảnh "slow

---

[4] Ở đây tôi dùng compiler của Dev-C++ 5.11. Các compiler khác có thể hơi khác.

motion" kiểu như thời gian trôi chậm hơn[5]. Điều quan trọng là ta cần đồng bộ trên mỗi khung hình, nghĩa là mỗi khung hình phải có đầy đủ hình ảnh Tom đuổi Jerry trong một trạng thái trung gian nào đó. Nếu không đồng bộ bạn có thể thấy tai, mắt, mũi họng của Tom và Jerry xuất hiện lộn xộn trong video. Ta cần vẽ trọn vẹn (đầy đủ) mỗi khung hình rồi mới hiển thị khung hình đó (kết xuất) chứ không kết xuất từng phần của mỗi khung hình. Để làm điều này ta sẽ không kết xuất trực tiếp từng phần ra màn hình (như cách Screen đã làm) mà ta cần một "màn hình ẩn" cho phép "kết xuất" từng phần lên đó và khi đã xong tất cả các phần ta sẽ hiển thị màn hình ẩn lên "màn hình thật". Đó là lí do mà ta có một module màn hình nữa gọi là BuffScreen[6]. Nơi dùng BuffScreen xem nó như là một Screen nghĩa là các dịch vụ của Screen đều có ở BuffScreen với cùng tên gọi và cách dùng. Tuy nhiên BuffScreen đặc biệt hơn Screen bình thường là những kết xuất sẽ không hiện lên màn hình cho đến khi nó được ra tín hiệu "hiển thị lên màn hình thật đi!". Ở đây ta thấy như vậy module trọng tâm của chương trình là Board không dùng Screen mà dùng BuffScreen để có được khả năng đồng bộ kết xuất. Bản thân BuffScreen sẽ dùng Screen để tương tác với màn hình thật. Về mặt kĩ thuật, BuffScreen sẽ dùng một mảng 2 chiều để chứa kết quả kết xuất (khung hình), ở đây là các kí tự. Sau đây là vài mô tả về BuffScreen:

- Cấu trúc `BUFF_SCREEN` mô tả một bộ đệm, bao gồm mảng 2 chiều các kí tự (`buff`) và kích thước rộng (`width`), cao (`height`) của bộ đệm (khung hình, màn hình ẩn).
- Thật sự thì nhiều chức năng của BuffScreen là mô phỏng lại chức năng tương ứng của Screen nên có cách dùng giống nhau và tốt nhất là cùng tên (tên hàm). Tuy nhiên C có những hạn chế kĩ thuật mà ta phải đặt tên khác nhau cho các hàm nên ta sẽ dùng tiếp vĩ ngữ `buffscr` cho BuffScreen (còn của Screen là `scr`)[7]. Chẳng hạn hàm `putcharbuffscr` của BuffScreen cung cấp cùng chức năng như `putcharscr` của Screen nên chúng có cùng prototype và cùng ý nghĩa. Tuy nhiên khác biệt là `putcharscr` (của Screen) sẽ xuất kí tự ra màn hình thật còn `putcharbuffscr` (của BuffScreen) sẽ đưa kí tự vào bộ đệm.
- Một khác biệt về logic quan trọng nữa của BuffScreen so với Screen là kích thước. Trong khi Screen có thể nói là có kích thước không

[5] Thật ra thì ta sẽ thấy hình giật, nghĩa là ta không thấy được chuyển động trơn tru, hay đúng hơn là không thấy hiệu ứng chuyển động mà thấy rõ các ảnh tĩnh tuần tự. Để có gây cảm giác chuyển động thì cần ít nhất 24 khung hình trên giây. Điều này liên quan đến khả năng lưu hình của mắt.

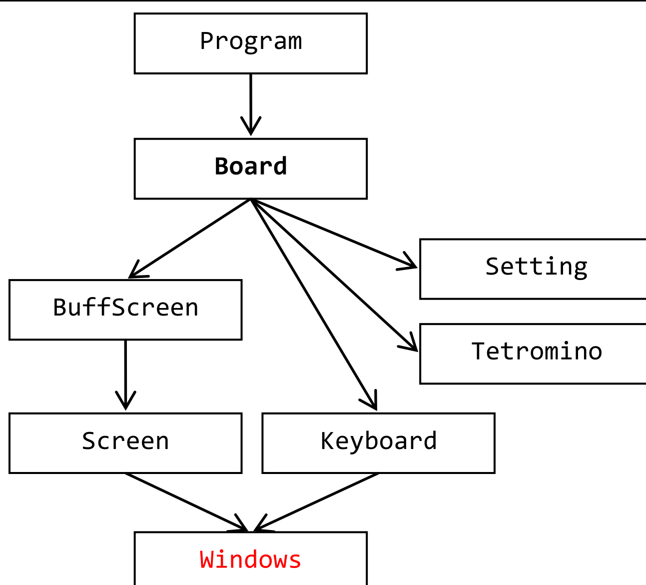
[6] Viết tắt của Buffered Screen, màn hình đệm hay màn hình ẩn.

[7] Thật sự thì các ngôn ngữ lập trình hướng đối tượng như C++, C#, Java mang lại một cách hiện thực hiệu quả, đẹp và hay hơn cho mối quan hệ giữa BuffScreen và Screen.

giới hạn thì BuffScreen có kích thước hữu hạn, không thay đổi và được đặt trước lúc "khởi động" nên hàm `initbuffscr` (của BuffScreen) có thêm 2 tham số kích thước còn hàm `initscr` (của Screen) thì không. Cũng lưu ý, do có kích thước hữu hạn nên những lời gọi kết xuất bên ngoài kích thước sẽ không có ý nghĩa (chỉ kết xuất trong kích thước khung hình, bên ngoài sẽ bị "xén bỏ").

- Ngoài hàm `clearbuffscr` cung cấp tiện ích xóa trắng màn hình BuffScreen mà Screen không có (vì không có ý nghĩa trên Screen do Screen không có giới hạn kích thước) thì khác biệt quan trọng của BuffScreen với Screen là hàm `outputbuffscr`. Đây chính là hàm thực thi tín hiệu "hiển thị lên màn hình thật đi!". Như bạn có thể đoán, hàm này dùng chức năng "kết xuất thật" của Screen.

Hình 4.9.1 – Sơ đồ module của chương trình

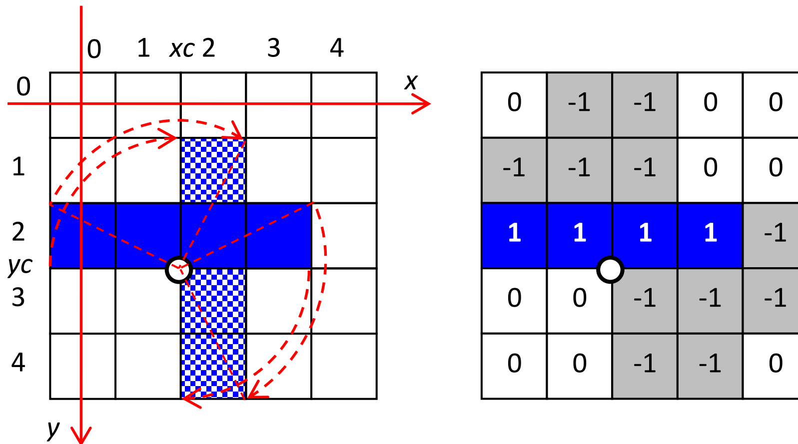


Tetromino là các đối tượng quan trọng nhất trong game này. Đó là các khối hình rơi xuống mà ta cần xếp trong Matrix. Có đúng 7 loại tetromino là *I*, *J*, *L*, *O*, *S*, *T*, *Z*[8] và ta sẽ "đóng gói" chúng trong module Tetromino. Kiểu cấu trúc `TETROMINO` giúp ta mô tả một tetromino. Kiểu này được khai báo trong file tiêu đề `tetromino.h` với các thành phần: `type` cho biết loại tetromino (một trong 7 kí tự I, J, …, Z ứng với 7 loại tetromino); mảng hai chiều `box` cho biết thông tin của tetromino; `width`, `height` cho biết kích thước của `box` và `xc`, `yc` cho biết tâm xoay của tetromino. Phần khó nhất trong việc quản lý các tetromino là quản lý việc xoay (quay) chúng. Các phiên bản game Tetris

[8] Tham khảo tài liệu nha.

khác nhau thường có các cách xoay khác nhau. Ở đây ta sẽ dùng cách xoay chuẩn trong Tetris Guideline là *SRS* (*Super Rotation System*). Thông tin của Tetris bao gồm khối hình và cách xoay có thể mô tả trong một hộp chữ nhật mà ta gọi là box. Hình 4.9.2 minh họa việc tổ chức thông tin này cho tetromino *I*.

Hình 4.9.2 – Hình minh họa việc tổ chức thông tin cho tetromino *I*



Mỗi tetromino là một khối hình gồm vài ô kề nhau. "Cấu hình chuẩn" của tetromino là cấu hình của khối hình khi tetromino xuất hiện. Mỗi tetromino cũng có một tâm xoay là điểm mà tetromino sẽ xoay quanh đó khi người chơi xoay nó. Lưu ý là người chơi chỉ được phép xoay tetromino mỗi lần một góc 90^0 theo chiều kim đồng hồ. Khi xoay tetromino, những ô mà khối hình của nó đi qua được gọi là các "ô xoay qua". Người chơi chỉ xoay được tetromino khi các ô xoay qua của nó không bị cấn (không có khối nào đó chiếm giữ). Như hình trên cho thấy, tetromino *I* có khối hình là đoạn thẳng 4 ô với cấu hình chuẩn nằm ngang. Tâm xoay là điểm có tọa độ (1.5, 2.5). Lưu ý là trong Toán ta hay dùng hệ trục tọa độ với chiều dương *y* hướng lên nhưng trong C thì hệ trục tọa độ với chiều dương *y* hướng xuống sẽ thuận tiện hơn. Các ô xoay qua là các ô tô xám (được xác định bằng cách xoay thử cấu hình chuẩn một góc 90^0 theo chiều kim đồng hồ). Ta thấy các ô liên quan đến tetromino *I* nằm trong một hình vuông có kích thước 5×5. Như vậy box của tetromino *I* là mảng 2 chiều kích thước 5×5 với giá trị các ô là 1, -1 hoặc 0 như hình. Giá trị 1 cho biết ô đó nằm trong khối hình, -1 cho biết ô đó là ô xoay qua và 0 cho biết ô đó không liên quan đến tetromino.

Thật ra trường `width`, `height` có thể được bỏ đi vì kích thước của box luôn là 5×5 với 7 loại tetromino hiện có. Tuy nhiên ta dùng chúng để dễ dàng mở rộng sau này (chẳng hạn ta có thể sẽ chế thêm các loại tetromino khác). Hơn nữa ta cũng dùng hằng tượng trưng `BOX_WIDTH_MAX` cho kích thước lớn nhất của box thay vì dùng cứng con số như 5. Hàm quan trọng

nhất (cũng là hàm public duy nhất) của module Tetromino là hàm `createtet` giúp tạo tetromino theo loại được cho. Nhiệm vụ của hàm này chính là điền giá trị cho các trường của tham số ra `tet`[9] theo thông tin của loại tetromino tương ứng. Ta cũng nhận xét rằng tâm xoay của các loại tetromino đều là (2, 2) trừ của *I* là (1.5, 2.5) và của *O* là (1.5, 1.5). Để tránh việc điền giá trị từng ô cho box rất dài dòng, Tetromino dùng hàm trợ thủ `_copybox` để chép giá trị cho `box` từ các biến mảng toàn cục tương ứng. Hiển nhiên các biến toàn cục này là riêng tư của Tetromino. Chẳng hạn biến mảng toàn cục cho box của tetromino *I* là `_Ibox` được khai báo với khởi gán trong `tetromino.cpp` như sau:

```
const int _Ibox[BOX_WIDTH_MAX][BOX_WIDTH_MAX] = {
                    {0, -1, -1, 0, 0},
                    {-1, -1, -1, 0, 0},
                    {1, 1, 1, 1, -1},
                    {0, 0, -1, -1, -1},
                    {0, 0, -1, -1, 0}};
```

Đây chính là thông tin box của tetromino *I* như được cho trong Hình 4.9.2. Từ khóa `const` đặt trước làm cho `_Ibox` không thể bị sửa dù là vô tình:)

Module Program cũng cực kì đơn giản như Bài 3.9. Có thể nói rằng nó y hệt Bài 3.9 nhưng thay Document bằng Board. Module Board là trọng tâm của chương trình này (như module Document là trọng tâm của chương trình ở Bài 3.9). Trong ứng dụng Char Paint ở Bài 3.9 thì hoạt động vẽ diễn ra trên bảng vẽ mà ta gọi là Document còn ở đây hoạt động game diễn ra trên khung cảnh mà ta gọi là Board. Một thành phần quan trọng trên Board đó là Matrix, nơi các tetromino rơi xuống và ta sẽ xếp chúng trên đó. Kích thước điển hình của Matrix là 10×20 nhưng Board cho phép đặt kích thước khác trong lời gọi `initboard` mà hiển nhiên là không được đặt chiều rộng quá `MAT_WIDTH_MAX` và chiều cao quá `MAT_HEIGHT_MAX`. Thật sự thì file tiêu đề `board.h` rất giống file tiêu đề `document.h` của Bài 3.9.

Điểm khác biệt quan trọng của module Board với module Document trong Bài 3.9 là ta đóng gói mọi thứ của Board vào trong một kiểu cấu trúc là `Board`, do đó ta chỉ cần dùng một biết toàn cục `_board` thay vì dùng rất nhiều biến toàn cục như trong Document. Kiểu `Board` có thành phần `matrix` là mảng nguyên 2 chiều để mô tả trạng thái các ô trong Matrix với giá trị 0 là ô trống và 1 là ô đã có "gạch". Thành phần `tet` (kiểu `TETROMINO`) cho biết tetromino đang rơi xuống trong Matrix và `nexttet` cho biết tetromino kế tiếp.

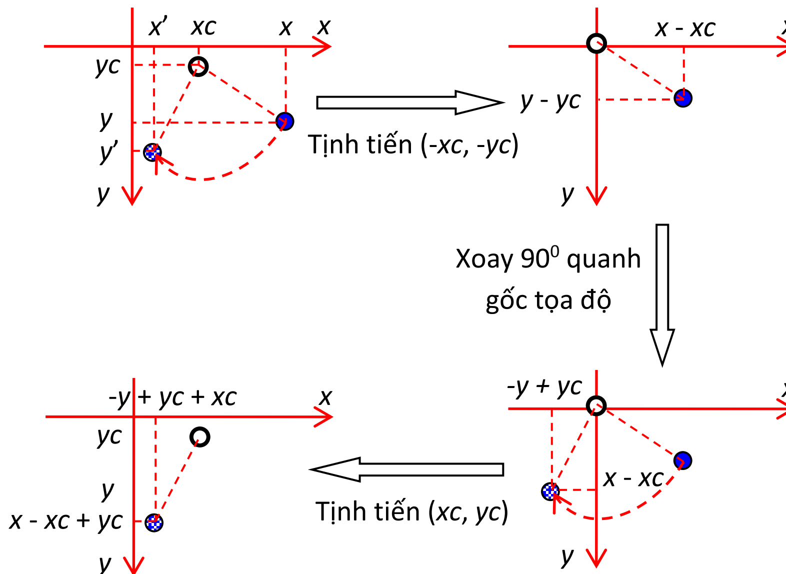
Để quản lý việc xoay tetromino, thành phần `tetdir` của kiểu `Board` cho biết hướng xoay hiện tại của tetromino. `tetdir` là 0 ứng với cấu hình chuẩn

---

[9] Ở đây ta chọn dùng tham số ra thay vì dùng giá trị trả về cho `TETROMINO` cần tạo một phần là do `TETROMINO` có kích thước khá lớn.

(không xoay), `tetdir` là 90 ứng với góc xoay 90^0 theo chiều kim đồng hồ so với cấu hình chuẩn, ... Do đó hàm xoay tetromino, `_rotate`, được viết rất đơn giản: tăng `tetdir` (của `_board`) thêm 90^0 nhưng nhớ là khi xoay đủ 360^0 thì tetromino đã quay lại cấu hình chuẩn (góc 0^0). Công việc khó hơn nằm ở hàm kiểm tra có xoay tetromino được hay không, hàm `_canrotate`. Để làm việc này ta xét các ô xoay qua của tetromino (các ô -1 trong box của tetromino) nếu có ô nào đó trên Matrix tương ứng với ô xoay qua mà đã có gạch thì không thể xoay tetromino do bị cấn. Vì thông tin trong box là thông tin của cấu hình chuẩn (góc 0^0) nên việc quan trọng ở đây là xác định tọa độ tương ứng của một ô khi xoay một góc `tetdir` (theo chiều kim đồng hồ) với tâm xoay là (`xc`, `yc`). Hàm `_map` nhận tọa độ một ô (`srow`, `scol`) và tính tọa độ ô tương ứng đó trong (`drow`, `dcol`). Ta cần làm Toán một chút để có thể viết hàm này. Hình sau đây minh họa trường hợp góc xoay là 90^0.

Hình 4.9.3 – Hình minh họa cách tính tọa độ tương ứng khi xoay góc 90^0



Để xoay điểm (*x*, *y*) góc 90^0 (theo chiều kim đồng hồ) quanh điểm (*xc*, *yc*) ta thực hiện tuần tự 3 bước sau:

(1) Tịnh tiến theo vector (-*xc*, -*yc*) để đưa tâm xoay về gốc tọa độ. Khi đó điểm (x, y) biến thành điểm (*x* – *xc*, *y* – *yc*).

(2) Xoay 90^0 quanh gốc tọa độ. Khi đó hoành độ sẽ thành tung độ mới và tung độ sẽ thành đối của tung độ mới (lưu ý là chiều dương trục tung hướng xuống) nên điểm (*x* – *xc*, *y* – *yc*) biến thành điểm (-(*y* – *yc*), *x* – *xc*) = (-*y* + *yc*, *x* – *xc*).

(3) Tịnh tiến theo vector (*xc*, *yc*) để đưa tâm về vị trí cũ. Khi đó điểm (-*y* + *yc*, *x* – *xc*) biến thành điểm (-*y* + *yc* + *xc*, *x* – *xc* + *yc*).

Tóm lại, điểm (*x*, *y*) sau khi xoay góc 90^0 quanh điểm (*xc*, *yc*) sẽ biến thành điểm (*x'*, *y'*) với *x'* = -*y* + *yc* + *xc* và *y'* = *x* – *xc* + *yc*. Như vậy đoạn mã tương ứng trong hàm `_map` sẽ là:

```
if(_board.tetdir == 90)
{
    drow = scol - _board.tet.xc + _board.tet.yc;
    dcol = -srow + _board.tet.yc + _board.tet.xc;
}
```

với (`scol`, `srow`) là (*x*, *y*) và (`dcol`, `drow`) là (*x'*, *y'*). Bạn tự phân tích cho các trường hợp còn lại nhé.

Việc di chuyển tetromino (tịnh tiến qua trái, phải hay xuống một ô) thì đơn giản hơn nhiều. Thành phần `tetrow`, `tetcol` của kiểu `Board` cho biết tọa độ hiện tại của góc trên, trái của tetromino (điểm (0, 0) trong box của tetromino). Do đó để di chuyển qua trái ta chỉ cần giảm `tetcol` một đơn vị (`tetcol--`), di chuyển qua phải thì tăng `tetcol` còn di chuyển xuống thì tăng `tetrow`. Việc xác định tetromino có thể di chuyển được hay không cũng khá dễ dàng. Hàm `_canmove` giúp kiểm tra xem có thể tịnh tiến tetromino theo vector (`drow`, `dcol`) được hay không. Do đó ta gọi `_canmove(0, -1)` để kiểm tra xem có thể di chuyển qua trái không hay `_canmove(1, 0)` để kiểm tra xem có thể di chuyển xuống dưới không. Hàm này đơn giản duyệt qua các ô khối hình của tetromino (các ô 1 trong box của tetromino) nếu có ô nào đó trên Matrix tương ứng với ô khối hình mà đã có gạch thì không thể di chuyển tetromino do bị cấn. Để tìm tọa độ ô tương ứng thì hàm này cũng dùng `_map`. Hơn nữa sau khi `_map` thì nhớ cộng thêm `tetrow`, `tetcol` do kết quả mà `_map` trả về tính theo gốc là góc trên, trái của tetromino.

Module Setting giúp module Board quản lý những việc liên quan đến các thiết lập/cấu hình của game như điểm số, level, tốc độ, … Module này hiện được viết rất đơn giản. Chẳng hạn hàm `score` tính điểm khi người chơi phá được `totalrow` dòng theo công thức rất đơn giản: mỗi dòng được 10 điểm và cộng thêm 5 điểm cho mỗi dòng nhiều hơn một dòng. Hay việc tăng tốc độ game mỗi khi người chơi qua level trong hàm `incrlevel` cũng khá đơn giản. Bạn cần tham khảo Tetris Guideline hay các nguồn khác để sửa lại tinh vi hơn.

Về mặt kĩ thuật ta đã dùng số ngẫu nhiên để phát sinh ngẫu nhiên tetromino kế tiếp. Hơn nữa ta cũng dùng kĩ thuật liên quan đến quản lý thời gian do tetromino trong game sẽ tự động di chuyển xuống (với tốc độ ngày càng nhanh theo level game). Bạn hãy đọc mã và tra cứu thêm để nắm rõ nhé. Lưu ý là, liên quan đến thời gian, tôi đã "cài" một "trick" trong game mà giúp cho người chơi có thể chơi game rất dễ dàng. Là gì vậy?

BÀI TẬP

Bt 4.9.1 Hãy phát triển để game Tetris ở bài học trên được hoàn chỉnh hơn:

- Thêm chức năng chơi lại từ đầu (New Game)
- Quản lý danh sách điểm cao (Highscores)
- Điều chỉnh cách tính điểm và tăng tốc độ qua các level tốt hơn
- Gỡ bỏ "trick" liên quan đến việc tạm dừng game (Pause)
- Thêm tính năng rơi nhanh và "soft drop"
- Thêm tính năng trọng lượng (gravity) với "chain reactions" khi phá gạch
- Thêm màu sắc cho game
- …

Bt 4.9.2 Các game thực tế không được viết như cách ta đã làm ở bài học trên, chúng thường được phát triển dựa trên các công nghệ game, công nghệ đồ họa, multimedia cao cấp hơn[10]. Tuy nhiên như ta đã thấy với "Ascii Art" trong Bài 1.7, công nghệ đơn giản cũng có thể tạo nên các loại hình nghệ thuật/giải trí cao cấp[11]. "Viết game", cũng như các loại hình nghệ thuật/giải trí khác, đôi khi chỉ cần một ý tưởng độc đáo là cuốn hút người chơi/người thưởng thức. Tương tự kĩ thuật viết game Tetris trong bài học trên, hãy viết các game đơn giản như[12]: Pong, Pac-Man, Đua xe, Snake, Caro, Dò mìn, …

Bt 4.9.3 (Mở rộng[13]) Bạn chắc đã dùng và quen thuộc với các máy tính cầm tay (Calculator) khi học phổ thông. So với các thế hệ máy ban đầu thì các Calculator ngày nay thực sự là các thiết bị tính toán tiện dụng, mạnh mẽ với nhiều chức năng. Hãy vận dụng các công cụ và kĩ thuật mà bạn đã được học cho đến giờ để viết một chương trình C mô phỏng các Calculator này, chẳng hạn như máy Casio fx-570ES PLUS. (*Gợi ý*: dot matrix display, các hàm của thư viện chuẩn, các hàm của môi trường thực thi (Windows), mảng 1 chiều, mảng 2 chiều, các kiểu dữ liệu cơ bản, chuỗi, lặp, đệ qui, các thuật toán giải tích số, tổ chức module, … là các thứ mà bạn có thể dùng đến. Hãy tìm kiếm và tra cứu. Hãy sáng tạo trong thiết kế giao diện và tương tác người dùng.)

Thời gian làm từ 3 ngày đến 3 tuần tùy theo thời gian … nghỉ ngơi. Nếu hoàn thành được khoảng 50% (so với máy thực như Casio fx-570ES PLUS) thì bạn đủ công lực để luyện tiếp Quyển II. Nếu không bạn nên luyện lại Quyển I cho kĩ. (Làm xong, nếu được, hãy gởi cho tôi một bản để mọi người có thể học hỏi.) *Bài tập này thay cho lời kết*!

---

[10] Ta có thể lại gặp nhau trong chủ đề "Lập trình game".

[11] Chất liệu/vật liệu/kỹ thuật đơn sơ cũng có thể tạo nên những tác phẩm nghệ thuật đặc sắc.

[12] Đây đều là các game đơn giản nổi tiếng. Bạn hãy chơi thử để cảm nhận game, sau đó tra cứu, tìm hiểu về game trước khi bắt tay viết nó nhé.

[13] Yêu cầu của các bài tập được đánh dấu "Mở rộng" là khó nhưng chúng xứng đáng để bạn bỏ công làm: bạn sẽ được một chương trình rất thú vị và công lực lập trình của bạn cũng tăng nhanh đáng kể.

PHỤ LỤC A.1
CÀI ĐẶT CÔNG CỤ LẬP TRÌNH C

Bạn chắc chắn cần có máy tính để lập trình C. Tuy nhiên có máy tính thôi chưa đủ, bạn *cần phải có công cụ lập trình được cài đặt trên máy*. Đây là một hệ thống phần mềm giúp bạn thực hiện các công đoạn của việc lập trình. Hệ thống này được gọi là *môi trường phát triển tích hợp* (Integrated Development Environment), viết tắt là *IDE*. Như tên gọi, IDE bao gồm nhiều thành phần mà quan trọng nhất là: *trình soạn thảo mã nguồn* (source code editor) phục vụ cho việc gõ mã C, *trình biên dịch* (compiler) biên dịch mã C thành mã thực thi và *trình gỡ lỗi* (debugger) hỗ trợ việc tìm và loại bỏ lỗi trong chương trình.

Tóm lại *bạn cần IDE để có thể lập trình* (và học lập trình). Có nhiều IDE khác nhau mà bạn có thể dùng. Nếu máy của bạn đã được cài đặt một IDE nào đó thì bạn đã có thể bắt đầu việc học lập trình. Nếu không bạn nên chọn cài 2 IDE là: *Dev-C++* và/hoặc *Microsoft Visual Studio*. Dev-C++ là một IDE nhỏ, gọn phù hợp cho việc học lập trình C/C++ còn Microsoft Visual Studio là một IDE chuyên nghiệp, hoành tráng, phù hợp cho việc viết các phần mềm lớn. Dev-C++ thì hoàn toàn miễn phí còn Microsoft Visual Studio thì có nhiều bản, có bản giá vài ngàn đô dùng cho các doanh nghiệp nhưng cũng có bản miễn phí dùng cho sinh viên hay lập trình viên không chuyên. Trong tài liệu này bạn nên dùng Dev-C++ khi luyện Tầng 1, 2, 3 để khỏi bỡ ngỡ và dùng Microsoft Visual Studio khi luyện Tầng 4 để quen với môi trường phát triển chuyên nghiệp. Cũng lưu ý là còn nhiều IDE tuyệt vời khác mà bạn nên trải nghiệm, nhưng trước mắt, bạn nên quen thuộc với 2 IDE ở trên.

Việc cài đặt các IDE khá đơn giản. Dễ nhất, bạn nhờ một ai đó có kinh nghiệm cài đặt giùm[1]. Nếu không bạn cũng có thể cài đặt Dev-C++ và Microsoft Visual Studio (bản miễn phí) theo tài liệu hướng dẫn đính kèm với Phụ lục này. Hoặc bạn cũng có thể làm theo các tài liệu hướng dẫn cài đặt khác (như các video hay tutorial trên mạng).

---

[1] Như nhờ thầy cô khi bạn là học sinh, sinh viên hoặc nhờ con cháu khi bạn lớn tuổi hoặc nhờ soái ca khi bạn là bánh bèo vô dụng:), …. Chắc chắn có thể nhờ các lập trình viên nhưng những game thủ cũng cài đặt được dễ dàng.

PHỤ LỤC A.2
GÕ VÀ CHẠY CHƯƠNG TRÌNH C

Thuật ngữ “gõ và chạy chương trình” ám chỉ 3 bước chính:

- *Bước* 1: Gõ mã chương trình.
- *Bước* 2: Biên dịch chương trình.
- *Bước* 3: Chạy chương trình.

Trước khi gõ mã chương trình, bạn cần phải tạo Project, tạo File (*Bước* 0). Sau khi chạy chương trình, bạn đóng vai trò *người dùng chương trình* để quan sát kết quả chạy chương trình hay tương tác với chương trình như nhập dữ liệu cho chương trình, ... (*Bước* 4). Hơn nữa chương trình có thể có *lỗi biên dịch*, nghĩa là chương trình bạn gõ không hợp lệ (xem Bài 1.7). Khi đó bạn cần phải biết chỗ bị lỗi và sửa lỗi (*Bước* 2’).

Tất cả các bước trên đều được các IDE hỗ trợ mà thao tác cụ thể phụ thuộc vào IDE mà bạn dùng. Bạn xem tài liệu hướng dẫn đính kèm với Phụ lục này để biết cách thực hiện trên Dev-C++ và Microsoft Visual Studio. Hoặc bạn cũng có thể làm theo các tài liệu hướng dẫn khác (như các video hay tutorial trên mạng).

PHỤ LỤC A.3
LỊCH SỬ C

Ngôn ngữ lập trình C do *Dennis Ritchie* phát triển từ năm 1969 đến năm 1973 tại Bell Labs. Từ khi xuất hiện (khoảng năm 1972) đến nay, C đã trải qua nhiều lần thay đổi và hoàn thiện với các phiên bản C khác nhau. Các cột mốc đáng kể là:

- K&R C: bản C đầu tiên được đặc tả bằng tài liệu nổi tiếng "*The C Programming Language*" (edition 1) do Brian Kernighan và Dennis Ritchie viết năm 1978. Phiên bản C này còn được gọi là *C gốc*.
- ANSI C: bản C được đặc tả bằng tài liệu "*The C Programming Language*" (edition 2) và được ANSI, ISO chuẩn hóa. Phiên bản C này còn được gọi là *C chuẩn* hay C89 và được hầu hết các compiler C hỗ trợ.
- C99: ISO chuẩn hóa C một lần nữa vào năm 1999 với việc bổ sung thêm vài đặc trưng mới cho C. Đa số compiler C hỗ trợ phiên bản C này (đầy đủ hoặc một phần).
- C11: Một bản chuẩn hóa nữa (vào năm 2011) với việc bổ sung thêm nhiều đặc trưng khác cho C. Một vài compiler C hỗ trợ phiên bản C này.
- Ngoài ra có một phiên bản C được chuẩn hóa cho các thiết bị nhúng (embedded system) được gọi là *Embedded C*.

C là ngôn ngữ lập trình được dùng phổ biến nhất từ trước đến giờ cho nhiều mục đích/loại phần mềm khác nhau trên nhiều hệ máy/kiến trúc máy khác nhau. Đặc biệt C rất gần gũi với kiến trúc máy nên được dùng để viết các *phần mềm hệ thống* (system software), các chương trình đặc biệt cần tốc độ hay khả năng can thiệp cấp thấp vào kiến trúc máy/phần cứng máy. Vì lí do này, đôi khi C còn được gọi là *ngôn ngữ lập trình cấp thấp* (low-level programming language) so với các *ngôn ngữ lập trình cấp cao* (high-level) như C#, Java, Python, SQL, …[1] C cũng được dùng để viết các *phần mềm ứng dụng* (application software), tuy nhiên, ở lãnh vực này thì các *ngôn ngữ lập trình hướng đối tượng* (như C++, Objective-C, C#, Java, …) tỏ ra phù hợp hơn nhất là cho các phần mềm thương mại.

---

[1] Chữ thấp/cao ở đây ám chỉ đến sự gần gũi, khả năng can thiệp vào máy. Theo nghĩa này thì *mã máy* (machine code) và *hợp ngữ* (assembly) có cấp thấp nhất, C có cấp cao hơn một chút (medium-level) còn Perl, Python, … có cấp cao nhất (very high-level), còn Prolog, SQL, … thì thuộc hàng siêu cấp:)

C là ngôn ngữ lập trình theo *phong cách* (paradigm) *thủ tục* (imperative, procedural) với chương trình *có cấu trúc* (structured). Đây là phong cách lập trình cơ bản nhất, phù hợp nhất với kiến trúc máy và cũng là cách tự nhiên nhất để mô tả thuật toán hay các bước thực hiện của một công việc. Do đó việc học lập trình trước hết nên bắt đầu bằng việc học lập trình C. Điều này không chỉ giúp bạn có cách thức lập trình, cách thức tư duy, giải quyết vấn đề phù hợp mà còn giúp bạn dễ dàng nắm được các phong cách lập trình, tư duy khác.

Đặc biệt C ảnh hưởng đến rất nhiều ngôn ngữ lập trình quan trọng ra đời sau này như: C++, Objective-C, C#, Java, JavaScript, Perl, PHP, Python, … Điều đó cũng làm cho việc học lập trình trước hết nên bắt đầu bằng việc học lập trình C bởi nó sẽ giúp bạn dễ dàng học các ngôn ngữ lập trình quan trọng khác. Về mặt này, có thể xem C như là ngôn ngữ Latin[2] của thế giới các ngôn ngữ lập trình.

Bạn không cần thuộc tất cả những điều ở trên. Thật ra chúng đều là những lời sáo rỗng (nếu bạn không thực sự trải nghiệm để biết/hiểu chúng). Có lẽ bạn chỉ cần nhớ *cha đẻ của ngôn ngữ C là Dennis Ritchie* để có thể hiểu trọn vẹn Bài 1.1 là đủ:) Tuy nhiên việc biết những chuyện bên lề, những chuyện tầm phào (như chữ C được lấy theo tên ngôn ngữ lập trình B hay thậm chí có một ngôn ngữ lập trình là ngôn ngữ lập trình D?:)), những chuyện thú vị … giúp những cuộc trò chuyện C thêm rôm rả và giúp bạn có cảm hứng học C hơn. Bạn hãy tham khảo thêm các bài viết về C nhé[3]!

---

[2] Chứ không phải là tiếng Anh hay tiếng Hoa nha:) Vị trí này thuộc về các ngôn ngữ lập trình khác.

[3] Có thể xuất phát từ bài viết “C (programming language)” trên trang Wikipedia: https://en.wikipedia.org/wiki/C_(programming_language)

PHỤ LỤC A.4
CẤU HÌNH CỬA SỔ CONSOLE

Sau khi "gõ và chạy chương trình", bạn thường đóng vai *người dùng chương trình*, quan sát kết quả hay tương tác với chương trình như nhập dữ liệu, ... để kiểm tra chương trình. Có rất nhiều dạng/loại chương trình khác nhau tùy theo mức độ đơn giản/phức tạp hay công nghệ/nền tảng/môi trường/phần cứng mà chương trình thực thi. Trong tài liệu này ta sẽ viết các chương trình ở dạng đơn giản là *chương trình Console*. Đây là dạng chương trình mà kết xuất và tương tác của chương trình với người dùng diễn ra trên một "cửa sổ" đơn giản gọi là *cửa sổ Console*. Trên cửa sổ này, tất cả các tương tác (nhập/xuất) đều thông qua *kí tự* và *chuỗi*. Như vậy chương trình Console sử dụng công nghệ tương tác, đồ họa, giao diện đơn giản. Ngược lại, *chương trình GUI* (Graphical User Interface[1]) sử dụng công nghệ tương tác, đồ họa, giao diện cao cấp và phức tạp hơn[2].

Cấu hình cửa sổ Console một cách phù hợp sẽ giúp người dùng sử dụng chương trình Console tốt hơn. Tuy nhiên, các môi trường (hệ điều hành) thường quản lý và cho phép cấu hình cửa sổ Console theo cách khác nhau. Bạn xem tài liệu hướng dẫn đính kèm với Phụ lục này để biết cách cấu hình cửa sổ Console trên Windows. Hoặc bạn cũng có thể xem các tài liệu hướng dẫn khác (như các video hay tutorial trên mạng).

---

[1] https://en.wikipedia.org/wiki/Graphical_user_interface

[2] Để so sánh với *GUI* thì dạng chương trình Console còn được gọi là *CUI* (viết tắt của từ tiếng Anh Command-line User Interface hay Console User Interface hay Character User Interface hay viết tắt của từ tiếng Việt *CÙI BẮP*:))
https://en.wikipedia.org/wiki/Command-line_interface

PHỤ LỤC A.5
TỪ VỰNG C

Tập các kí hiệu được phép dùng trong chương trình C được gọi là *bộ kí tự C*. Nói chung, đó là các kí tự thông thường gõ được từ bàn phím[1]. Ở mức thấp nhất, một chương trình C là một dãy các kí tự trong bộ kí tự C. Ở mức cao hơn, các kí tự được gom lại thành các đơn vị gọi là *từ*[2]. Giữa các từ thường có kí hiệu phân cách (như kí tự khoảng trắng) hoặc không cần kí hiệu phân cách nếu dễ dàng xác định "ranh giới" giữa các từ. Các từ trong C được gọi là *token* và có 6 loại token (*từ loại*) trong C:

1) *Từ khóa* (Keyword): là những từ đặc biệt, được C dành riêng, gồm[3]:

```
char     short     int      long    float
double   unsigned  signed   extern  static
const    void      struct   enum    union

if       else      switch   case    default
for      do        while    break   continue
goto     return    typedef  sizeof
```

   Các từ khóa được chia ra 2 nhóm: các từ khóa liên quan đến dữ liệu (nhóm trên) và các từ khóa liên quan đến lệnh (nhóm dưới).

2) *Định danh* (Identifier): là từ dùng để đặt tên như *tên biến*, *tên hàm*, *tên kiểu dữ liệu*. Định danh gồm một hoặc nhiều kí tự là *chữ cái* (letter) hoặc *chữ số* (digit), trong đó kí tự đầu tiên là chữ cái. Kí tự gạch dưới (_) được xem là chữ cái. Lưu ý, C phân biệt chữ hoa chữ thường nên x1, X1 là hai định danh khác nhau. Hơn nữa định danh không được trùng với từ khóa. Ví dụ:
   - Các định danh hợp lệ: `main`, `Main`, `printf`, `tong`, `Double`, `_1x`, `dien_tich`
   - Các định danh không hợp lệ: `tổng`[4], `US$`, `double`, `1x`, `dien-tich`, `dien tich`

---

[1] Tùy phiên bản C và compiler mà bộ kí tự có khác nhau. Một số compiler chấp nhận bộ kí tự Unicode, cho phép cả các kí tự có dấu tiếng Việt.

[2] Thật ra, trước khi tiến hành việc gom các kí tự thành từ, compiler sẽ thực hiện một quá trình gọi là *tiền xử lý* (preprocessing). Ta sẽ tìm hiểu quá trình này sau.

[3] Danh sách này chỉ gồm các từ khóa hay gặp nhất. Để đỡ rối, các từ khóa rất ít dùng thì không được liệt kê.

[4] Một số compiler chấp nhận định danh này và các định danh tiếng Việt khác (hay các ngôn ngữ khác). Điều này phụ thuộc vào bộ kí tự C mà compiler chấp nhận.

3) *Hằng số* (Constant), gồm:
   - *Hằng số nguyên*: mô tả một số nguyên cụ thể, được viết theo cơ số 10 như `0`, `100`, …[5] hoặc cơ số 16, bắt đầu bằng tiền tố `0x` hay `0X`, như `0x100`, `0x1a`, `0x1A`, … (xem Bài 1.7)[6].
   - *Hằng kí tự*: mô tả một kí tự cụ thể, với cách viết `'<kí tự>'`, như `'a'`, `'A'`, `'\n'`, `'\0'`, `'\x16'`, … Lưu ý, hằng kí tự thật ra là một hằng số, nó mô tả một số nguyên cố định, là mã ASCII của kí tự tương ứng (xem Bài 1.7).
   - *Hằng số thực*: mô tả một số thực cụ thể, được viết theo kiểu thập phân thông thường như `10.25`, `10.0`, `10.`, `.10`, … hay theo kiểu khoa hoặc như `1e-6` (cho số thực 1×10^{-6} = 0.000001), `0.001e4` (cho số thực 0.001×10^{4} = 10.0), …
4) *Hằng chuỗi* (String literal): mô tả một chuỗi cố định, với cách viết `"<các kí tự của chuỗi>"`, như `"Hello"`, `"Hello world!"`, `"Hi\tthere\n"`, …. (xem Bài 1.7).
5) *Toán tử* (Operator): là các kí tự mô tả phép toán như `+ - * / % = ++ ==` … (xem Phụ lục A.6).
6) *Dấu câu* (Separator): là các kí tự phân cách các phần của chương trình như `{ } ( ) , ;` …

Khoảng trắng (white space) là dãy các *kí tự trắng* (space, tab, xuống dòng). Chúng không là token của C nhưng giúp phân cách các token và *định dạng* chương trình cho dễ đọc. Lưu ý, hằng chuỗi được phép chứa kí tự space và nó có ý nghĩa (chuỗi xuất ra tương ứng sẽ có khoảng trắng). Kí tự tab và xuống dòng được mô tả bằng *chuỗi thoát* `\t` và `\n` tương ứng trong hằng chuỗi (xem Bài 1.7).

Ghi chú (comment) là chuỗi kí tự được đặt trong cặp dấu `/*` và `*/`. Ghi chú dạng này có thể trải dài trên nhiều dòng. Một cách ghi chú khác, trên một dòng là đặt chuỗi ghi chú sau dấu `//`[7]. Ghi chú được lập trình viên dùng để chú thích thêm các thông tin cho chương trình. Chúng không có ý nghĩa với compiler: chúng được compiler xem là khoảng trắng và được bỏ qua.

---

[5] Lưu ý: `-100` được xem gồm 2 token là `-` (toán tử) và `100` (hằng số nguyên).
[6] Thật ra C cũng cho phép mô tả hằng số nguyên bằng cơ số 8 nhưng ít được dùng.
[7] Ghi chú dạng này (trên một dòng với `//`) có nguồn gốc từ C++.

PHỤ LỤC A.6
CÁC TOÁN TỬ C

Bảng sau đây liệt kê danh sách các *toán tử* C cùng với độ ưu tiên và tính kết hợp của chúng. Để khỏi rối, danh sách này chỉ gồm các toán tử mà bạn được học trong Quyển 1 này. Danh sách đầy đủ tất cả các toán tử của C sẽ được cho trong Quyển 2.

Bảng A.6.1 – Danh sách các toán tử C cùng với độ ưu tiên và tính kết hợp

<table>
<tr><th>Độ ưu tiên</th><th>Toán tử</th><th>Thao tác</th><th>Tính kết hợp</th><th>Bài học</th></tr>
<tr><td rowspan="4">1</td><td>()</td><td>Gọi hàm</td><td rowspan="4">Trái</td><td>Bài 3.3</td></tr>
<tr><td>[]</td><td>Truy cập phần tử mảng</td><td>Bài 4.1</td></tr>
<tr><td>.</td><td>Truy cập thành phần cấu trúc</td><td>Bài 4.5</td></tr>
<tr><td>++ --</td><td>Tăng, giảm (hậu tố)</td><td>Bài 2.2</td></tr>
<tr><td rowspan="6">2</td><td>!</td><td>Phủ định luận lý</td><td rowspan="6">Phải</td><td>Bài 2.1</td></tr>
<tr><td>++ --</td><td>Tăng, giảm (tiền tố)</td><td>Bài 2.2</td></tr>
<tr><td>+ -</td><td>Giữ giá trị, lấy đối</td><td>Bài 2.1</td></tr>
<tr><td>(type)</td><td>Ép kiểu</td><td>Bài 3.5</td></tr>
<tr><td>sizeof</td><td>Lấy kích thước kiểu</td><td>Bài 3.5</td></tr>
<tr><td>&</td><td>Lấy địa chỉ</td><td>Mở rộng 2.1</td></tr>
<tr><td>3</td><td>* / %</td><td>Nhân, chia, chia lấy dư</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>4</td><td>+ -</td><td>Cộng, trừ</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>5</td><td>< <= > >=</td><td>So sánh nhỏ hơn, nhỏ hơn hoặc bằng, lớn hơn, lớn hơn hoặc bằng</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>6</td><td>== !=</td><td>So sánh bằng, khác</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>7</td><td>&&</td><td>Và luận lý</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>8</td><td>||</td><td>Hoặc luận lý</td><td>Trái</td><td>Bài 2.1</td></tr>
<tr><td>9</td><td>?:</td><td>Điều kiện</td><td>Phải</td><td>Bài 2.6</td></tr>
<tr><td>10</td><td>= += -= *= /= %=</td><td>Gán đơn, gán kép</td><td>Phải</td><td>Bài 2.2</td></tr>
<tr><td>11</td><td>,</td><td>Lượng giá dãy</td><td>Trái</td><td>Bài 2.6</td></tr>
</table>

Độ ưu tiên xác định thứ tự thực hiện các toán tử: các toán tử có độ ưu tiên cao hơn sẽ được thực hiện trước. Khi cùng độ ưu tiên, *tính kết hợp* sẽ xác định việc thực hiện các toán tử là từ trái qua phải (kết hợp Trái) hay từ

phải qua trái (kết hợp Phải) (xem Bài 2.1). Ở bảng trên, các toán tử chung một nhóm có độ ưu tiên như nhau, các toán tử ở nhóm trên (nhóm nhỏ) thì có độ ưu tiên cao hơn các toán tử ở nhóm dưới (nhóm lớn). Chẳng hạn, toán tử [ ] và toán tử chấm (.) có độ ưu tiên như nhau và cao nhất, toán tử phẩy (,) có độ ưu tiên thấp nhất.

Sau đây là một vài điểm về các toán tử C mà bạn nên lưu ý:

- C luôn ưu tiên thực hiện các toán tử trong cặp ngoặc tròn trước.
- Hầu hết các toán tử một ngôi có độ ưu tiên cao hơn các toán tử hai ngôi
- Các toán tử hậu tố (nhóm 1) thì có độ ưu tiên cao hơn các toán tử tiền tố (nhóm 2) và các toán tử tiền tố thì có độ ưu tiên cao hơn các toán tử trung tố (các nhóm còn lại).
- Chỉ có 3 nhóm toán tử có tính kết hợp phải là nhóm 2 (các toán tử một ngôi tiền tố), nhóm 9 (toán tử điều kiện) và nhóm 10 (các toán tử gán). Các nhóm toán tử còn lại đều có tính kết hợp trái.
- Các toán tử gọi hàm, gán (đơn/kép) và tăng/giảm là các toán tử có hiệu ứng lề (xem Bài 2.2).
- Ngoại trừ các toán tử &&, ||, ?: và toán tử phẩy (,), C không xác định thứ tự lượng giá các toán hạng[1]. Thứ tự này do compiler quyết định và có thể ảnh hưởng đến việc lượng giá các biểu thức có hiệu ứng lề (xem Mở rộng 2.2).
- Toán tử luận lý và (&&), hoặc (||) được lượng giá tắt (xem Bài 2.6).
- Vài toán tử có cùng kí hiệu nhưng xác định các phép toán khác nhau tùy theo ngữ cảnh[2]. Chẳng hạn:
    - Kí hiệu +, - có thể mô tả cho toán tử giữ hay lấy đối giá trị (như trong +3, -2.5) và cũng có thể mô tả cho toán tử cộng hay trừ (như trong 3 + 2, 3.5 – 2.5). Để phân biệt, trường hợp đầu được gọi là toán tử cộng, trừ một ngôi và trường hợp sau là toán tử cộng, trừ hai ngôi.
    - Toán tử một ngôi tăng, giảm (++, --) có thể được viết theo cách tiền tố (viết trước toán hạng) hoặc hậu tố (viết sau toán hạng) và được lượng giá theo hai cách khác nhau (xem Bài 2.2). Để phân biệt, trường hợp đầu được gọi là toán tử tăng, giảm tiền tố và trường hợp sau là toán tử tăng, giảm hậu tố.
    - Các toán tử số học (+, -, *, /, %) và các toán tử so sánh (<, <=, >, >=, ==, !=) đều có phiên bản cho số nguyên và số thực riêng, tùy theo kiểu của toán hạng (xem Bài 2.1).

[1] Điều này không liên quan gì đến tính kết hợp của các toán tử.

[2] Đây là hiện tượng "*lạm dụng kí hiệu*" mà ta đã thấy trong Bài 2.1.

PHỤ LỤC A.7
DEBUG CHƯƠNG TRÌNH

Chương trình có *lỗi biên dịch* là chương trình không hợp lệ. Compiler không biên dịch được những chương trình như vậy ra mã thực thi nên chúng không chạy được. Sửa các lỗi biên dịch là công việc khá dễ dàng dưới sự trợ giúp của IDE (xem Phụ lục A2). Tuy nhiên chương trình không có lỗi biên dịch (và do đó chạy được) vẫn có thể có *bug*, các lỗi hay bất thường khi chạy (xem Bài 1.6). Các bug này thường để lại các hậu quả "khủng khiếp" cho chương trình. Hơn nữa chúng thường khó phát hiện và khó sửa hay loại bỏ. Do đó kĩ năng debug, tìm và loại/sửa bug, là một kĩ năng thực hành rất quan trọng mà mọi lập trình viên đều phải thành thạo. Có rất nhiều cách thức, chiến lược debug khác nhau mà theo nghĩa rộng, bao gồm cả *kiểm tra*/*kiểm thử phần mềm* (Software verification/Software testing). Ở mức đơn giản nhất ta phải sử dụng thành thạo *debugger*, công cụ mà IDE cung cấp để hỗ trợ cho việc debug. Bạn xem tài liệu hướng dẫn đính kèm với Phụ lục này để biết cách dùng debugger của Dev-C++ và Microsoft Visual Studio. Hoặc bạn cũng có thể xem các tài liệu hướng dẫn khác (như các video hay tutorial trên mạng).

PHỤ LỤC A.8
THAO TÁC TRÊN TẬP TIN VĂN BẢN

Tập tin văn bản (Text file) là tập tin lưu trữ *chuỗi*, tức là dãy kí tự (xem Bài 4.7). Chính xác hơn, nó được gọi là tập tin *thuần văn bản* (plain text) vì dữ liệu của nó chỉ là văn bản mà không có các dữ liệu định dạng như font chữ, màu sắc, kích thước, … hay các đối tượng đồ họa như ảnh, bảng biểu, … đi kèm. Cách gọi này giúp phân biệt với dạng tập tin chứa *tài liệu văn bản* (document) như Word, … Các tập tin văn bản thường có phần mở rộng là *.txt*[1] và được xử lý bằng các *trình soạn thảo văn bản* (Text editor). Trên Windows ta có thể dùng phần mềm *Notepad*. Tuy nhiên có nhiều phần mềm xử lý tập tin văn bản tốt hơn với nhiều chức năng hơn mà ta nên dùng, nhất là với các lập trình viên, để có thể xử lý tập tin văn bản chuyên sâu hơn. Một phần mềm như vậy là *Notepad++* mà rất đáng bỏ công để cài đặt và dùng thành thạo. Bạn xem tài liệu hướng dẫn đính kèm với Phụ lục này để biết cách cài đặt Notepad++ và cách thực hiện các thao tác đơn giản trên tập tin văn bản bằng Notepad và Notepad++. Hoặc bạn cũng có thể xem các tài liệu hướng dẫn khác (như các video hay tutorial trên mạng).

[1] Tập tin văn bản xuất hiện hầu khắp nơi:) và có thể có phần mở rộng khác, chẳng hạn các tập tin CSV (có phần mở rộng là .csv), các dạng tập tin văn bản đánh dấu (có phần mở rộng là .html, .xml, …) hay các tập tin chứa mã nguồn của các ngôn ngữ lập trình (chẳng hạn mã nguồn C/C++ với phần mở rộng là .c, .cpp, .h), …

PHỤ LỤC A.9
TẠO TẬP TIN CSV

Tập tin CSV (Comma-Separated Values) là tập tin (thuần) văn bản lưu trữ dữ liệu của một *bảng tính* (xem Mở rộng 4.7). Vì là tập tin văn bản nên ta có thể dùng các *trình soạn thảo văn bản* (Text editor) như Notepad, Notepad++, … để tạo tập tin này. Tuy nhiên, bảng tính thường được tạo và nhập liệu bằng các *phần mềm xử lý bảng tính* (Spreadsheet) chuyên nghiệp như Microsoft Excel, … Các phần mềm này đều lưu trữ bảng tính bằng tập tin có định dạng riêng như .xls, .xlsx, … mà chúng thường khá phức tạp và không phải là dạng tập tin văn bản. Dù vậy, hầu hết các phần mềm xử lý bảng tính đều cho phép lưu trữ bảng tính ở dạng tập tin CSV. Đây cũng là cách để trao đổi dữ liệu giữa các phần mềm xử lý bảng tính với nhau và/hoặc với các phần mềm xử lý dữ liệu khác. Bạn xem tài liệu hướng dẫn đính kèm với Phụ lục này để biết cách tạo bảng tính và lưu trữ ở dạng tập tin CSV bằng Microsoft Excel. Hoặc bạn cũng có thể xem các tài liệu hướng dẫn khác (như các video hay tutorial trên mạng).

TÀI LIỆU THAM KHẢO

1. Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*, 2nd edition. Prentice Hall, 1988.
2. Eric S. Roberts. *The Art and Science of C: A Library Based Introduction to Computer Science*. Addison-Wesley, 1994.
3. K. N. King. *C Programming: A Mordern Approach*, 2nd edition. W. W. Norton & Company, 2008.
4. Paul Deitel, Harvey Deitel. *C: How to Program*, 7th edition. Prentice Hall, 2012.
5. Trần Đan Thư, Nguyễn Thanh Phương, Đinh Bá Tiến, Trần Minh Triết. *Nhập môn Lập trình*. NXB Khoa học và Kỹ thuật, 2011.